14:06:33 OCA PAD INITIATION - PROJECT HEADER INFORMATION 05/11/89

Active

Project #: E-21-T22 Cost share #: E-21-330 Rev #: 0
Center # : R6583-T22 Center shr #: F6583-T22 OCA file #: 128
Work type : RES
Contract#: F30602-88-D-0025-0022 Mod #: Document : DO
Prime #: Contract entity: GTRC

Subprojects ? : N
Main project #:

Project unit: EE Unit code: 02.010.118
Project director(s):
PARIS D T EE (404)894-2902

Sponsor/division names: AIR FORCE / GRIFFISS AFB, NY
Sponsor/division codes: 104 / 023

Award period: 890417 to 900416 (performance) 900516 (reports)

| Sponsor amount | New this change | Total to date |
|---|---|---|
| Contract value | 99,693.00 | 99,693.00 |
| Funded | 45,000.00 | 45,000.00 |
| Cost sharing amount | | 5,000.00 |

Does subcontracting plan apply ?: Y

Title: NEURAL NETWORK COMMUNICATIONS PROCESSOR

PROJECT ADMINISTRATION DATA

OCA contact: Brian J. Lindberg 894-4820

Sponsor technical contact

RICHARD N. SMITH

DEPARTMENT OF THE AIR FORCE
ROME AIR DEVELOPMENT CENTER/DCCD
GRIFFISS AFB, NY 13441-5700

Sponsor issuing office

GERARD J. BROWN/PKRM
(315)330-2417
ROME AIR DEVELOPMENT CENTER
DIRECTORATE OF CONTRACTING (PKRM)
GRIFFISS AFB, NY 13441-5700

Security class (U,C,S,TS) : U ONR resident rep. is ACO (Y/N): Y
Defense priority rating : DO-A7 GOV'T supplemental sheet
Equipment title vests with: Sponsor GIT
NONE PROPOSED OR ANTICIPATED.


Administrative comments -
DELIVERY ORDER PARTIALLY FUNDS TASK C-9-2402 (STANFORD UNIVERSITY) THROUGH SEPTEMBER 30, 1989.

GEORGIA INSTITUTE OF TECHNOLOGY
OFFICE OF CONTRACT ADMINISTRATION

NOTICE OF PROJECT CLOSEOUT

---

Closeout Notice Date 01/14/91

Project No. E-21-T22_____ Center No. R6583-T22_____

Project Director JOY E B_____ School/Lab ELEC ENGR_____

Sponsor AIR FORCE/GRIFFISS AFB, NY_____

Contract/Grant No. F30602-88-D-0025-0022_____ Contract Entity GTRC

Prime Contract No. _____

Title NEURAL NETWORK COMMUNICATIONS PROCESSOR_____

Effective Completion Date 900630 (Performance) 900730 (Reports)

---

| Closeout Actions Required: | Y/N | Date Submitted |
|---|---|---|
| Final Invoice or Copy of Final Invoice | Y | _____ |
| Final Report of Inventions and/or Subcontracts | Y | _____ |
| Government Property Inventory & Related Certificate | N | _____ |
| Classified Material Certificate | Y | _____ |
| Release and Assignment | Y | _____ |
| Other SUBCONTRACTS CLOSEOUT_____ | Y | _____ |

Comments_____

_____

Subproject Under Main Project No. _____

Continues Project No. _____

Distribution Required:

| | |
|---|---|
| Project Director | Y |
| Administrative Network Representative | Y |
| GTRI Accounting/Grants and Contracts | Y |
| Procurement/Supply Services | Y |
| Research Property Managment | Y |
| Research Security Services | N |
| Reports Coordinator (OCA) | Y |
| GTRC | Y |
| Project File | Y |
| Other _____ | N |
| _____ | N |

NOTE: Final Patent Questionnaire sent to PDPI.

# Neural Networks Communication Processor*

## Monthly Status Report
## October 1989

**Technical Accomplishments:** Work concentrated primarily on noise reduction. Implementation of both feedforward and feedback neural networks in the nonlinear adaptive noise cancelling structure was accomplished. Backpropagation and the on line recursive algorithm were used successfully to adapt the nonlinear filter of the adaptive noise canceller. Coding and testing of the neural network vector quantization implementation continued.

**Research Participants:**

| Name | Title | Percent of Time |
|---|---|---|
| Bernard Widrow | Professor | 20.0% |
| Stephen Piche | Research Assistant | 50.0% |
| Joice DeBolt | Secretary | 10.0% |

**Trips:** None

**Presentations:** None

**Publications:** None

* This work is supported by subcontract E-21-T22-S1 between Stanford University and Georgia Institute of Technology. This subcontract is supported under prime contract F30602-88-D-0025 between Georgia Institute of Technology and Rome Air Development.

# Neural Networks Communication Processor*

## Monthly Status Report
## November 1989

**Technical Accomplishments:** Work concentrated on a calculation of the computational complexity of the on-line recursive algorithm. The computational complexity was found to compare favorably under certain circumstances to the complexity of the backpropagation through time algorithm. In addition to the work on computational complexity, the second quarterly report was compiled during the month of November.

**Research Participants:**

| Name | Title | Percent of Time |
|---|---|---|
| Bernard Widrow | Professor | 20.0% |
| Stephen Piche | Research Assistant | 50.0% |
| Joice DeBolt | Secretary | 10.0% |

**Trips:** None

**Presentations:** None

**Publications:** None

* This work is supported by subcontract E-21-T22-S1 between Stanford University and Georgia Institute of Technology. This subcontract is supported under prime contract F30602-88-D-0025 between Georgia Institute of Technology and Rome Air Development.

# Neural Networks Communication Processor*

## Monthly Status Report
## April 1990

**Technical Accomplishments**: Work continued on a paper on the subject of adapting neural networks with feedback. The paper will discuss both the on-line backpropagation-through-time algorithm and on-line recursive algorithm.

**Research Participants:**

| Name | Title | Percent of Time |
|---|---|---|
| Bernard Widrow | Professor | 20.0% |
| Stephen Piche | Research Assistant | 50.0% |
| Joice DeBolt | Secretary | 10.0% |

**Trips**: None

**Presentations**: None

**Publications**: None

* This work is supported by subcontract E-21-T22-S1 between Stanford University and Georgia Institute of Technology. This subcontract is supported under prime contract F30602-88-D-0025 between Georgia Institute of Technology and Rome Air Development.

# Neural Networks Communication Processor*

## Monthly Status Report
## May 1990

**Technical Accomplishments**: Work continued on a paper on the subject of adapting neural networks with feedback. The paper will discuss both the on-line backpropagation-through-time algorithm and on-line recursive algorithm.

**Research Participants**:

| Name | Title | Percent of Time |
|---|---|---|
| Bernard Widrow | Professor | 20.0% |
| Stephen Piche | Research Assistant | 50.0% |
| Joice DeBolt | Secretary | 10.0% |

**Trips**: None

**Presentations**: None

**Publications**: None

* This work is supported by subcontract E-21-T22-S1 between Stanford University and Georgia Institute of Technology. This subcontract is supported under prime contract F30602-88-D-0025 between Georgia Institute of Technology and Rome Air Development.

# Neural Networks Communication Processor*

## Monthly Status Report
## June 1990

**Technical Accomplishments**: Work concentrated on generalization of the on-line backpropagation-through-time and on-line recursive algorithms. These algorithms form the basis of paper currently being written.

**Research Participants**:

| Name | Title | Percent of Time |
|---|---|---|
| Bernard Widrow | Professor | 20.0% |
| Stephen Piche | Research Assistant | 50.0% |
| Joice DeBolt | Secretary | 10.0% |

**Trips**: Stephen Piche and Bernard Widrow both attended the International Joint Conference on Neural Networks in San Diego.

**Presentations**: None

**Publications**: None

* This work is supported by subcontract E-21-T22-S1 between Stanford University and Georgia Institute of Technology. This subcontract is supported under prime contract F30602-88-D-0025 between Georgia Institute of Technology and Rome Air Development.

# Neural Networks Communication Processor*

## Monthly Status Report
## July 1990

**Technical Accomplishments:** Implementation of a couple of neural network systems with feedback was started. These systems are designed to illustrate the usefulness of the recently developed generalized on-line learning algorithms.

**Research Participants:**

| Name | Title | Percent of Time |
|---|---|---|
| Bernard Widrow | Professor | 20.0% |
| Stephen Piche | Research Assistant | 50.0% |
| Joice DeBolt | Secretary | 10.0% |

**Trips:** None

**Presentations:** None

**Publications:** None

* This work is supported by subcontract E-21-T22-S1 between Stanford University and Georgia Institute of Technology. This subcontract is supported under prime contract F30602-88-D-0025 between Georgia Institute of Technology and Rome Air Development.

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: MAY-JUN '88

| | | | |
|---|---|---|---|
| | CURRENT QUARTER FUNDING | | $0.00 |
| | CURRENT QUARTER EXPENDITURES | | $0.00 |
| | CONTRACT CEILING | | $4,200,000.00 |
| | FUNDING TO DATE | - | $0.00 |
| * | PENDING COMMITMENTS | - | $766,000.00 |
| | AVAILABLE FUNDING | | $3,434,000.00 |
| | FUNDING TO DATE | | $0.00 |
| | YTD EXPENDITURES | - | $0.00 |
| | OUTSTANDING EXPENDITURES | | $0.00 |

| | | |
|---|---|---|
| * | C-8-2120 WESTINGHOUSE/BEAUDET | $56,000.00 |
| | C-8-2129 RENSSELAER/DAS | $100,000.00 |
| | E-8-7066 UNIV OF PENN/STEINBERG | $100,000.00 |
| | E-8-7124 BOSTON COLLEGE/McFADDEN | $35,000.00 |
| | E-8-7125 BRANDEIS UNIV/HENCHMAN | $23,000.00 |
| | E-8-7126 PENN STATE/CASTLEMAN | $22,000.00 |
| | A-8-1631 UNIV OF PENN/STEINBERG | $100,000.00 |
| | B-8-3617 GA WASHINGTON UNIV/MELTZER | $100,000.00 |
| | B-8-3618 GA WASHINGTON UNIV/BERKOVICH | $100,000.00 |
| | C-8-2492 GA TECH/SMITH | $50,000.00 |
| | A-8-1203 GA TECH/HUGHES | $80,000.00 |
| | TOTAL PENDING | $766,000.00 |

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: OCT-DEC '88

| | | | |
|---|---|---|---|
| CURRENT QUARTER FUNDING | | | $120,834.00 |
| DO # 0004 | $66,680 | | |
| 0006 | $54,154 | | |
| | $120,834 | | |
| CURRENT QUARTER EXPENDITURES | | | $28,740.82 |
| CONTRACT CEILING | | | $4,200,000.00 |
| FUNDING TO DATE | | - | $818,868.00 |
| * PENDING COMMITMENTS | | - | $784,729.00 |
| AVAILABLE FUNDING | | | $2,596,403.00 |
| FUNDING TO DATE | | | $818,868.00 |
| YTD EXPENDITURES | | - | $28,740.82 |
| OUTSTANDING EXPENDITURES | | | $790,127.18 |

| | | | |
|---|---|---|---|
| * DO # 0001 | INCREMENTAL FUNDING | | $90,729.00 |
| 0007 | INCREMENTAL FUNDING | | $20,000.00 |
| C-8-2400 | STATE UNIV OF NY/FAM | | $95,000.00 |
| C-8-2402 | RENSSELAER/SAULNER | | $100,000.00 |
| B-9-3592 | UNIV OF CA/DAVIS/LEVITT | | $60,000.00 |
| N-9-5514 | SOHAR INC./HECHT | | $50,000.00 |
| C-9-2015 | NCS/O'NEAL | | $100,000.00 |
| A-9-1120 | HITEC, INC./KAZAKOS | | $75,000.00 |
| E-9-7057 | UNIV OF TX/ARLINGTON/FUNG | | $40,000.00 |
| E-9-7093 | MONTANA STATE/JOHNSON | | $34,000.00 |
| S-9-7552 | ALFRED UNIV/SYNDER | | $20,000.00 |
| C-9-2404 | STANFORD UNIV/WIDROW | | $100,000.00 |
| | TOTAL PENDING | | $784,729.00 |

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JAN-MAR '89

| | | | |
|---|---|---|---|
| CURRENT QUARTER FUNDING | | | $574,457.00 |
| DO # | 0001 | $90,729 | |
| | 0011 | $75,000 | |
| | 0012 | $75,000 | |
| | 0013 | $59,989 | |
| | 0014 | $49,989 | |
| | 0015 | $70,000 | |
| | 0016 | $43,750 | |
| | 0017 | $30,000 | |
| | 0018 | $22,000 | |
| | 0019 | $38,000 | |
| | 0020 | $20,000 | |
| | | $574,457 | |
| CURRENT QUARTER EXPENDITURES | | | $86,324.15 |
| CONTRACT CEILING | | | $4,200,000.00 |
| FUNDING TO DATE | | | - $1,393,325.00 |
| * PENDING COMMITMENTS | | | - $594,651.00 |
| AVAILABLE FUNDING | | | $2,212,024.00 |
| FUNDING TO DATE | | | $1,393,325.00 |
| YTD EXPENDITURES | | | - $115,064.97 |
| OUTSTANDING EXPENDITURES | | | $1,278,260.03 |

| | | | |
|---|---|---|---|
| * DO # | 0007 | INCREMENTAL FUNDING | $20,000.00 |
| | 0011 | INCREMENTAL FUNDING | $19,568.00 |
| | 0012 | INCREMENTAL FUNDING | $24,700.00 |
| | 0015 | INCREMENTAL FUNDING | $29,783.00 |
| | 0016 | INCREMENTAL FUNDING | $31,250.00 |
| | 0017 | INCREMENTAL FUNDING | $10,000.00 |
| | 0018 | INCREMENTAL FUNDING | $12,000.00 |
| | 0019 | INCREMENTAL FUNDING | $12,000.00 |
| | C-8-2404 | STANFORD UNIV/WIDROW | $100,000.00 |
| | N-9-5732 | GRIFFIN | $25,000.00 |
| | A-9-1476 | BOWDOIN COLLEGE/CHONACKY | $20,350.00 |
| | E-9-7110 | UNIV OF LOWELL/SALES | $50,000.00 |
| | S-9-7559 | UNIV OF MICHIGAN/ROBINSON | $20,000.00 |
| | B-9-3621 | SRI/LUNT | $20,000.00 |
| | N-9-5308 | KAMAN SCIENCES | $100,000.00 |
| | E-9-7119 | DARTMOUTH COLLEGE/CRANE | $100,000.00 |
| | | TOTAL PENDING | $594,651.00 |

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: APR-JUN '89

| | | | |
|---|---|---|---|
| | CURRENT QUARTER FUNDING | | $160,350.00 |
| | DO # 0021 | $25,000 | |
| | 0022 | $45,000 | |
| | 0023 | $20,350 | |
| | 0024 | $50,000 | |
| | 0025 | $20,000 | |
| | | $160,350 | |
| | CURRENT QUARTER EXPENDITURES | | $318,963.82 |
| | CONTRACT CEILING | | $4,200,000.00 |
| | FUNDING TO DATE | - | $1,553,675.00 |
| * | PENDING COMMITMENTS | - | $718,994.00 |
| | AVAILABLE FUNDING | | $1,927,331.00 |
| | FUNDING TO DATE | | $1,553,675.00 |
| | YTD EXPENDITURES | - | $434,028.79 |
| | OUTSTANDING EXPENDITURES | | $1,119,646.21 |

| | | | |
|---|---|---|---|
| * | DO # 0007 | INCREMENTAL FUNDING | $20,000.00 |
| | 0011 | INCREMENTAL FUNDING | $19,568.00 |
| | 0012 | INCREMENTAL FUNDING | $24,700.00 |
| | 0015 | INCREMENTAL FUNDING | $29,783.00 |
| | 0016 | INCREMENTAL FUNDING | $31,250.00 |
| | 0017 | INCREMENTAL FUNDING | $10,000.00 |
| | 0018 | INCREMENTAL FUNDING | $12,000.00 |
| | 0019 | INCREMENTAL FUNDING | $12,000.00 |
| | 0022 | INCREMENTAL FUNDING | $54,693.00 |
| | B-9-3621 | SRI/LUNT | $20,000.00 |
| | N-9-5308 | KAMAN SCIENCES | $100,000.00 |
| | E-9-7119 | DARTMOUTH COLLEGE/CRANE | $100,000.00 |
| | N-9-5740 | CHRISTIANSON | $15,000.00 |
| | N-9-5317 | UNIV OF CO/NORGARD | $50,000.00 |
| | S-9-7625 | UNIV OF CA/DAVIS/KOWELL | $20,000.00 |
| | N-9-5314 | KAMAN SCIENCES | $100,000.00 |
| | N-9-5315 | KAMAN SCIENCES | $100,000.00 |
| | | TOTAL PENDING | $718,994.00 |

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JUL-SEP '89

| | | | |
|---|---|---|---|
| CURRENT QUARTER FUNDING | | | $476,000.00 |
| DO # | 0017 | $10,000 | |
| | 0026 | $15,000 | |
| | 0027 | $20,000 | |
| | 0028 | $50,000 | |
| | 0029 | $40,000 | |
| | 0030 | $30,000 | |
| | 0031 | $20,000 | |
| | 0032 | $66,000 | |
| | 0033 | $70,000 | |
| | 0034 | $85,000 | |
| | 0035 | $70,000 | |
| | | $476,000 | |
| CURRENT QUARTER EXPENDITURES | | | $415,422.69 |
| CONTRACT CEILING | | | $4,200,000.00 |
| FUNDING TO DATE | | | - $2,029,675.00 |
| * PENDING COMMITMENTS | | | - $253,994.00 |
| AVAILABLE FUNDING | | | $1,916,331.00 |
| FUNDING TO DATE | | | $2,029,675.00 |
| YTD EXPENDITURES | | | - $849,451.48 |
| OUTSTANDING EXPENDITURES | | | $1,180,223.52 |

| | | | |
|---|---|---|---|
| * DO # | 0007 | INCREMENTAL FUNDING | $20,000.00 |
| | 0011 | INCREMENTAL FUNDING | $19,568.00 |
| | 0012 | INCREMENTAL FUNDING | $24,700.00 |
| | 0015 | INCREMENTAL FUNDING | $29,783.00 |
| | 0016 | INCREMENTAL FUNDING | $31,250.00 |
| | 0018 | INCREMENTAL FUNDING | $12,000.00 |
| | 0019 | INCREMENTAL FUNDING | $12,000.00 |
| | 0022 | INCREMENTAL FUNDING | $54,693.00 |
| | N-0-5703 | UNIV OF SOUTHERN FLA/WILSON | $50,000.00 |
| | | TOTAL PENDING | $253,994.00 |

EJ1-122

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: OCT-DEC '89

| | | | | |
|---|---|---|---|---|
| CURRENT QUARTER FUNDING | | | | $292,994.00 |
| DO # | 0001 | $9,000 | C-8-2129 | |
| | 0011 | $19,568 | C-8-2400 | |
| | 0012 | $24,700 | C-8-2402 | |
| | 0015 | $29,783 | C-9-2015 | |
| | 0016 | $31,250 | A-9-1120 | |
| | 0018 | $12,000 | E-9-7093 | |
| | 0019 | $62,000 | C-9-2109 | |
| | 0022 | $54,693 | C-9-2404 | |
| | 0028 | $50,000 | N-9-5308 | |
| | | $292,994 | | |

| | | |
|---|---|---|
| CURRENT QUARTER EXPENDITURES | | $286,691.16 |
| CONTRACT CEILING | | $4,200,000.00 |
| FUNDING TO DATE | - | $2,322,669.00 |
| * PENDING COMMITMENTS | - | $595,000.00 |
| AVAILABLE FUNDING | | $1,282,331.00 |
| FUNDING TO DATE | | $2,322,669.00 |
| YTD EXPENDITURES | - | $1,136,142.64 |
| OUTSTANDING EXPENDITURES | | $1,186,526.36 |

| | | | | |
|---|---|---|---|---|
| * | DO # 0007 | S-8-7592 | INCREMENTAL FUNDING | $20,000.00 |
| | 0029 | E-9-7119 | INCREMENTAL FUNDING | $60,000.00 |
| | 0030 | N-9-5317 | INCREMENTAL FUNDING | $20,000.00 |
| | 0034 | N-9-5314 | INCREMENTAL FUNDING | $15,000.00 |
| | 0016 | N-9-5315 | INCREMENTAL FUNDING | $30,000.00 |
| | N-0-5703 | UNIV OF SOUTHERN FLA/WILSON | | $50,000.00 |
| | A-0-1102 | UNIV OF CA/SMOOT, BARBER, GT | | $100,000.00 |
| | P-0-6011 | NCSU/VANDERLUGT | | $100,000.00 |
| | C-0-2456 | NEW JERSEY INST/BAR-NESS | | $100,000.00 |
| | P-0-6014 | STEVENS INST/ZMUDA | | $100,000.00 |
| | TOTAL PENDING | | | $595,000.00 |

WAITING FOR PROPOSALS:

| | |
|---|---|
| P-0-6018 | UAH/CAULFIELD |
| P-0-6021 | GT/SUMNERS |
| P-0-6022 | CORNELL UNIV/TANG |
| B-0-3353 | ROCHESTER INST/LASKY |

CONTRACT FUNDS STATUS REPORT (DD FORM 1586)
CONTRACT NUMBER F30602-88-D-0025
QUARTER: JAN-MAR '90

| | | | | |
|---|---|---|---|---|
| CURRENT QUARTER FUNDING | | | | $114,301.00 |
| DO # | 0007 | $9,000 | S-8-7592 | |
| | 0029 | $19,568 | E-9-7119 | |
| | 0030 | $24,700 | N-9-5317 | |
| | 0036 | $29,783 | P-0-6014 | |
| | 0037 | $31,250 | P-0-6011 | |
| | | $114,301 | | |

| | | | |
|---|---|---|---|
| | CURRENT QUARTER EXPENDITURES | | $376,743.62 |
| | CONTRACT CEILING | | $4,200,000.00 |
| | FUNDING TO DATE | - | $2,436,970.00 |
| * | PENDING COMMITMENTS | - | $532,800.00 |
| | AVAILABLE FUNDING | | $1,230,230.00 |
| | FUNDING TO DATE | | $2,436,970.00 |
| | YTD EXPENDITURES | - | $1,512,886.26 |
| | OUTSTANDING EXPENDITURES | | $924,083.74 |

| | | | | |
|---|---|---|---|---|
| * | DO# | 0034 | N-9-5314 INCREMENTAL FUNDING | $15,000.00 |
| | | 0035 | N-9-5315 INCREMENTAL FUNDING | $30,000.00 |
| | | 0037 | P-0-6011 INCREMENTAL FUNDING | $10,000.00 |
| | N-0-5703 | | UNIV OF SOUTHERN FLA/WILSON | $50,000.00 |
| | A-0-1402 | | UNIV OF CA/SMOOT, BARBER, GT | $100,000.00 |
| | C-0-2456 | | NEW JERSEY INST/BAR-NESS | $100,000.00 |
| | P-0-6021 | | GT/SUMNERS | $100,000.00 |
| | P-0-6022 | | CORNELL UNIV/TANG | $30,800.00 |
| | B-0-3353 | | ROCHESTER INST/LASKY | $20,000.00 |
| | P-0-6018 | | UAH/CAULFIELD | $77,000.00 |
| | | | TOTAL PENDING | $532,800.00 |

WAITING FOR PROPOSALS:
P-0-6018 UAH/CAULFIELD
P-0-6021 GT/SUMNERS
P-0-6022 CORNELL UNIV/TANG
B-0-3353 ROCHESTER INST/LASKY

E-21-T22

# Neural Networks Communication Processor

First Quarterly Report
(April 1989 - July 1989)

Prepared by

B. Widrow, S. Piche

Information Systems Laboratory
Department of Electrical Engineering
Stanford University, Stanford, CA 94305

for

Georgia Institute of Technology
Subcontract No. E-21-T22-S1

and

Rome Air Development Center
Contract No. F30602-88-D-0025

# Table of Contents

## 1.0 Introduction

A survey of the field of communications has been conducted and several problems have been identified where neural network technology might offer solutions. The results of the survey are the subject of this report.

Recent advances in multilayer neural networks have been shown to be useful in fields such as control, pattern recognition and digital image processing. There is little reason to doubt that such methods could not be useful in the field of communication. With this in mind, this survey was conducted to identify problems in which the recent advances in multilayer neural networks could offer solutions to problems which could not be solved using linear single layer neural networks.

The results of the survey are broken down into the following two sections: Noise Reduction and Vector Quantization. Adaptive echo cancelling, multipath reduction and beamforming are all examples of adaptive noise reduction systems. These systems currently use time invariant linear single layer networks as the adaptive structure. Nonlinear multilayer networks may be advantageous for noise reduction in the presence of nonlinear noise. Time varying linear and nonlinear neural networks may be useful in systems with nonstationary noise. The first section of the results section is devoted to this subject.

Vector quantization is used for data compression. A vector quantizer is a system for mapping a sequence of continuous or discrete vectors into a digital sequence for communication over a digital channel. Current methods utilize adaptive methods; however, they lack backward error propagation. It may be possible to use neural networks adapted using backpropagation to solve this problem. The second section of the results section discusses this subject.

An additional section of this report contains two other possible applications of neural networks for communication applications: faster linear convergence and adaptive jammer direction identification. These applications have been separated from the

other two applications because they appear not to be as promising as the noise reduction and vector quantization applications. However, they do warrant a brief discussion and some further investigation.

The final section of the report contains a discussion of future work. Two appendices with preliminary noise reduction results have also been included.

## 2.0 Noise Reduction

Most applications of adaptive systems in communications depend upon adaptive noise reduction. As mentioned above, adaptive echo cancelling, multipath reduction and beamforming are all examples of adaptive noise reduction systems. Therefore, developments in adaptive noise reduction would have broad applications in the communications field.

Two methods of noise reduction are discussed in this section. The first method may be used to increase the signal to noise ratio when nonlinear noise is present in the signal. It is similar to the linear adaptive noise canceller which requires a noise reference input. The second method uses a time varying neural network for noise reduction. This method may be used when nonstationary linear or nonlinear noise is present in the signal.

### 2.1 Nonlinear Noise Reduction

Linear single layer neural networks have been used for noise reduction since the early 60's [1]. The structure of a linear noise reduction system is shown in Figure 1.

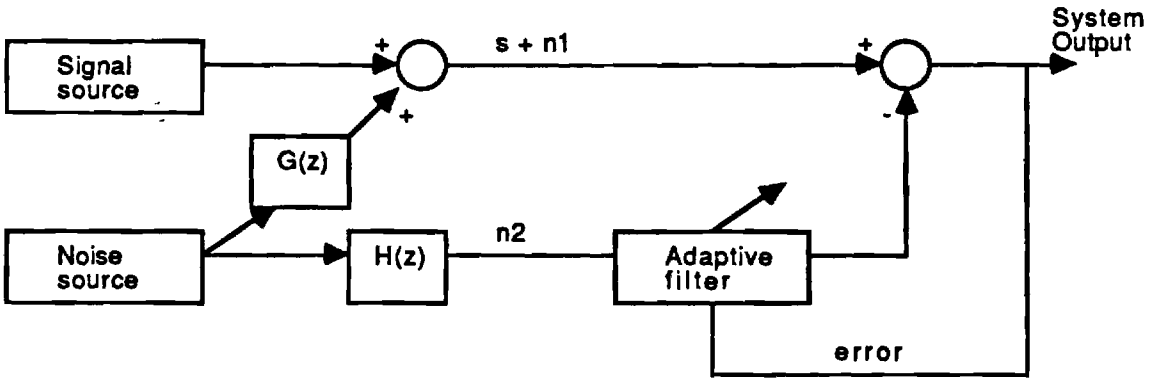


Figure 1: Adaptive noise-canceling concept

The signal with noise is input to one node and a noise reference is input to the other node. The system output is used as an error signal to adjust a linear adaptive filter using the LMS algorithm[1]. The output of the adaptive filter learns to emulate the noise present in the primary signal. This results in a significant reduction in noise at the output. This scheme has been used successfully in many noise reduction applications such as 60 Hz noise cancelation and echo cancelling. Modifications of this scheme have been used for adaptive beamforming.

The system shown in Figure 1 performs well when the noise introduced to the signal is linear. However, the noise introduced into the signal is not always linear. It is possible that the noise may be nonlinear or linear noise may pass through a nonlinear system before being added to the signal. In this case, it may be best to use a nonlinear adaptive filter for noise reduction. The structure of the nonlinear noise reduction filter is the same as the linear noise reduction filter except that the linear adaptive filter is replace with a nonlinear multilayer neural network. Figure 2 shows the structure of the nonlinear adaptive noise cancelling system.

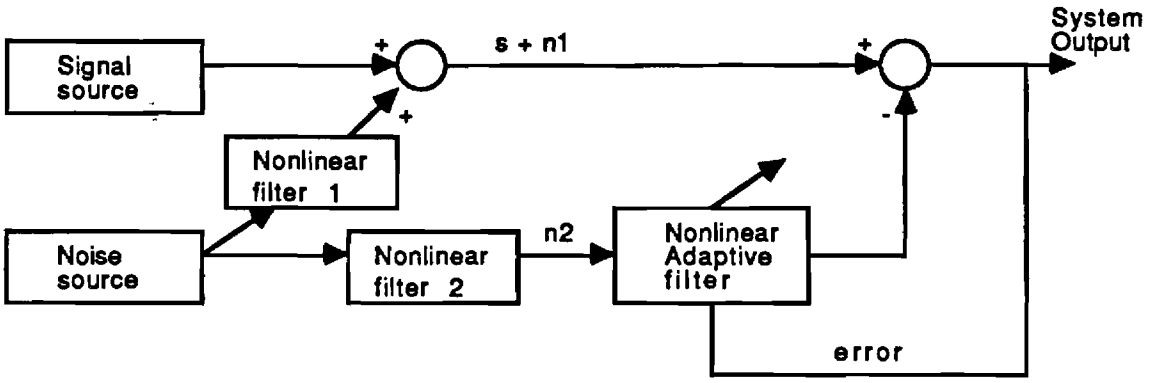


Figure 2: Nonlinear adaptive noise-cancelling concept

Preliminary investigations, included in Appendix A, show that improvements of signal to noise ratios are possible with a simple single layer nonlinear network. These results indicate that in certain cases a nonlinear network can reduce the signal to noise ratio and also suggests that the structure above can be used for noise reduction.

## 2.2 Time Varying Neural Networks for Noise Reduction

Given the autocorrelation of the input signal and the correlation between the desired signal and input signal, discrete Wiener filtering is used for optimal noise reduction when the process is stationary and uncorrelated over time. The assumption that the process is stationary and uncorrelated over time allows the optimal noise reduction filter to be time invariant. It is well known that the LMS algorithm will converge to the Wiener solution.

When the process is stationary and the signal is correlated over time, a Recursive Wiener filter results in optimal noise filtering. A Recursive Wiener filter uses not only the observed noise signal at each time step as input, it also uses the output of the filter at the previous time step. The Recursive Wiener filter is time variant because the output of the filter is nonstationary. This implies that the input at the next time step is nonstationary.

A recent advance in neural network training algorithms referred to as backpropagation through time may allow efficient and accurate training of the Recursive Wiener solution[2]. A preliminary investigation which is included in Appendix B indicates that

backpropagation through time can be used to find the Recursive Wiener solution.

## 3.0 Vector Quantization

Vector quantization has become an increasingly popular form of data compression[3]. Depending on the application, vector quantization can often decrease the amount of data required for transmission through a digital communication channel by an order of magnitude. It is possible that neural network vector quantizers could provide even better data compression.

Neural networks seem well suited for the vector quantization problem. Most vector quantization techniques, including the standard LBG vector quantizer, use some form of iterative training in a fashion similar to the training required for neural networks.

Tradition vector quantizers are comprised of an encoder, decoder and codebook as shown in the Figure 3. The codebook contains a set number of digital codes into which each vector must be encoded. After a vector has been encoded and sent through a communication channel, a decoder uses the codebook to reconstruct the original vector. The goal of vector quantization is to minimize the size of the codebook while also minimizing the distortion of the reconstructed vector.

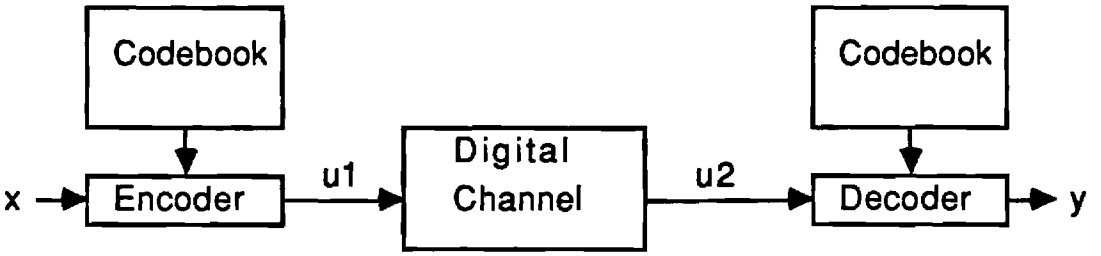


Figure 3: Traditional Vector Quantizer Structure

## 3.1 Tree Structure Vector Quantization

Due to design and implementation considerations, vector quantizers have varying structure. One form of vector quantizer (VQ), referred to as a tree structure VQ, achieves vector quantization by a series of successive binary decisions. These binary decisions separate the input space into various partitions which are represented by branches of the tree structure. A code out of the codebook is associated with each of the bottom nodes of the tree structure. This code is used to represents any vector in the input space which falls into the space partitioned to the associated node. For a binary tree with N levels, the total number of codewords is $2^N$.

The tree structure provides computational advantages over traditional structures. Since only one comparison is needed at each level, the computations required to search a tree grows with the logarithm of the codebook size. This a great savings over the fullsearch VQ which grows linearly with codebook size.

## 3.2 Neural Network Implementation

We are currently investigating using neural networks to perform the binary decisions of a tree structure vector quantizer. The structure of the neural network and connections between networks is the central issue in this investigation. We hope that by the next report we will have a better understanding of the required structure.

# 4.0 Faster Linear Convergence and Jammer Direction Identification

This section discusses two other possible applications of neural networks for communications: faster linear convergence and adaptive jammer direction identification. These applications have been separated from the other two applications because they initially appear not be as promising as the noise reduction and vector quantization applications.

## 4.1 Faster Linear Convergence

The single layer neural network has proven very useful in many communications applications. Any improvements in the linear neural network's convergence properties would undoubtedly result

in improvements in currently used adaptive communication applications.

Convergence of the LMS algorithm is very slow when the ratio of the largest eigenvalue of the input signal's autocorrelation matrix to the smallest eigenvalue of the matrix is large. It is hoped that under these conditions addition nodes and layers may improve convergence properties. The addition of nodes and layers to the linear neural network structure results in an error surface which is different from the original linear single layer error surface. It is possible that the shape of this error surface would allow faster convergence in cases where the ratio of eigenvalues is large.

Figure 5 illustrates the manner in which additional nodes and layers could be added to a linear single layer network. Figure 5.a shows the standard single layer linear structure. Figure 5.b shows the multilayer linear structure. The error surfaces of the two networks will be quite different since the first network has two weights and the second network has six weights.

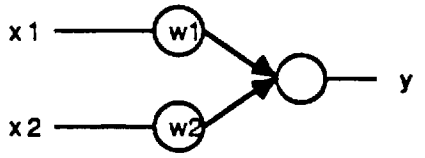


Figure 5.a: Single Layer

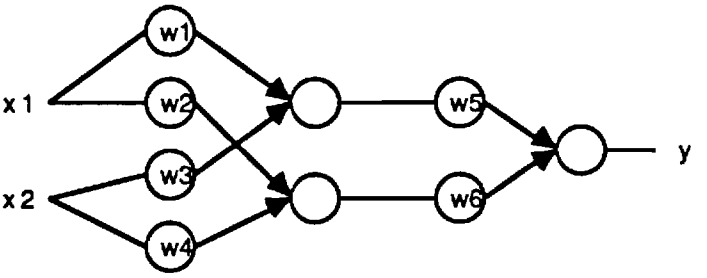


Figure 5.b: Multilayer Linear Network

## 4.2 Adaptive Jammer Direction Identification

Adaptive beamformers are used to receive a desired directional signal in the presence of an interference signal from an unknown direction. An adaptive filter is used to eliminate the unwanted interference signal from the desired signal. Adaptive beamformers such as the Frost and Griffiths-Jim adaptive beamformers operate by using the system output as an error signal for an adaptive filter or a

bank of adaptive filters[1]. They never explicitly identify the direction of the interference signal.

It may be possible to add a neural network to the adaptive beamformers to identify the direction of the interference signal. There exists a nonlinear relationship between the weights of the adaptive beamformer and the direction of the interference signal. A neural network could be used to learn this nonlinear relationship. The problem becomes more difficult when more interference signals are introduced. The neural network would be required to learn the mappings between the weights and the multiple directions.

## 5.0 Future Work

This report surveyed the field of communications and identified problems in which neural network technologies may offer solutions. Future work shall investigate the feasibility of various neural network methodologies for each of the problems identified in this report. The next quarterly report shall include the results of this study. It is hoped that after the feasibility study is complete, two problems can be chosen with concurrence of the RADC Task Engineer for an indepth study.

Appendix A: Nonlinear Noise Reduction

The results of using a nonlinear filter for noise reduction is discussed in this appendix. The nonlinear filter was used to recover the desired signal which is shown in Figure A.2 from the noisy signal which is shown in Figure A.3. The noisy signal is created by adding noise with a uniform distribution between 0.35 and -0.35 to the desired signal.

The structure of the neural network used for the nonlinear filter is shown in Figure A.1.b. The output of the nonlinear filter shall be compared to the linear filter shown in Figure A.1.a. The input to both the linear and nonlinear filter consisted of a window of 40 continuous samples of the noisy signal. Both filters were trained on ten test samples similar to the one shown in Figure A.3 using LMS. The desired response was the desired signal corresponding to the center of the input window. After training, the noisy signal shown in A.3 was presented to both filters. Figure A.4 shows the output of the nonlinear filter and Figure A.5 shows the output of the linear filter. These results show that the nonlinear filter performed better than the linear filter. In this case, the mean squared error of the nonlinear filter was 3db lower than the linear filter.

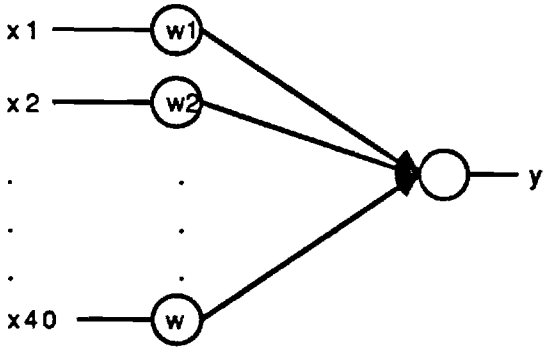


Figure A.1.a: Linear filter

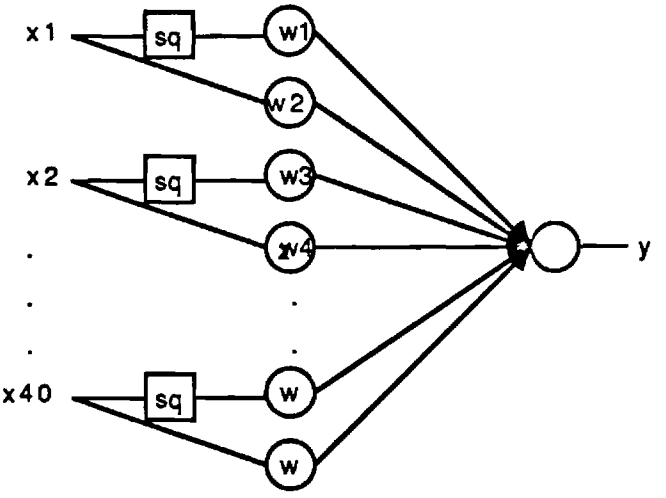


Figure A.2.b: Nonlinear filter

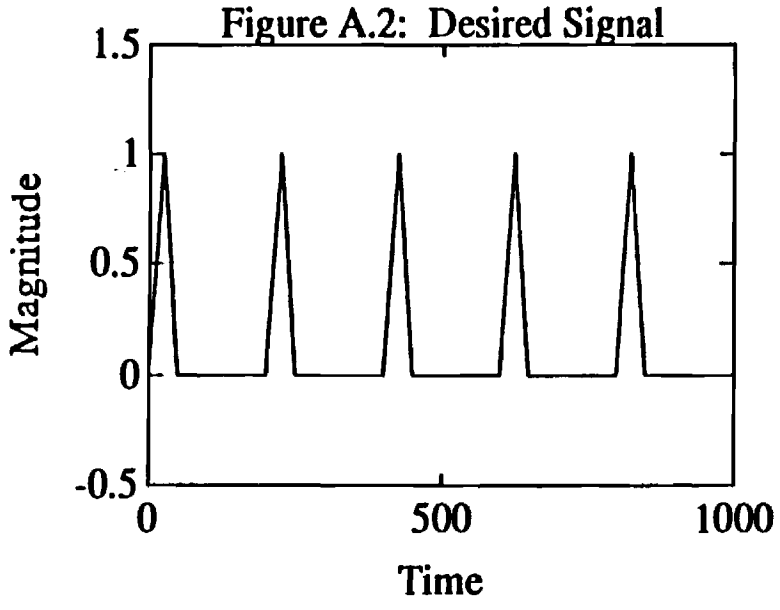
Figure A.2: Desired Signal
Magnitude
1.5
1
0.5
0
-0.5
0
500
1000
Time

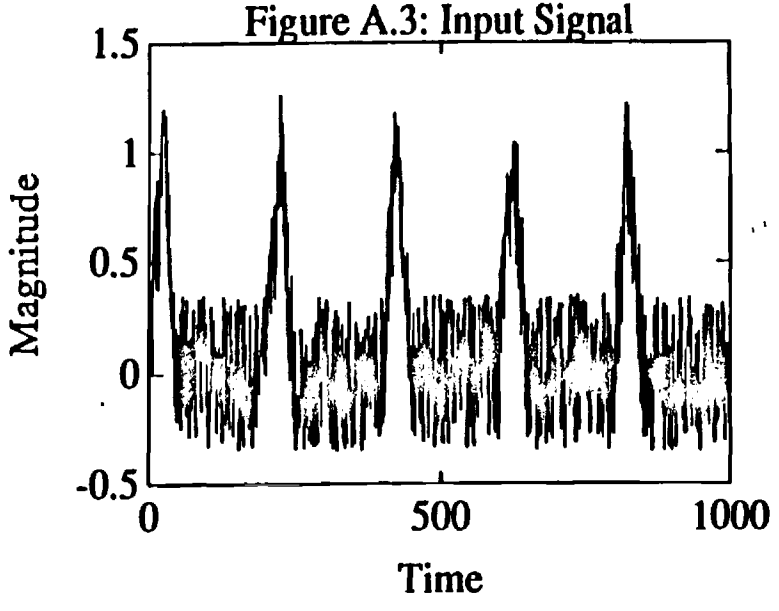
Figure A.3: Input Signal
Magnitude
1.5
1
0.5
0
-0.5
0
500
1000
Time

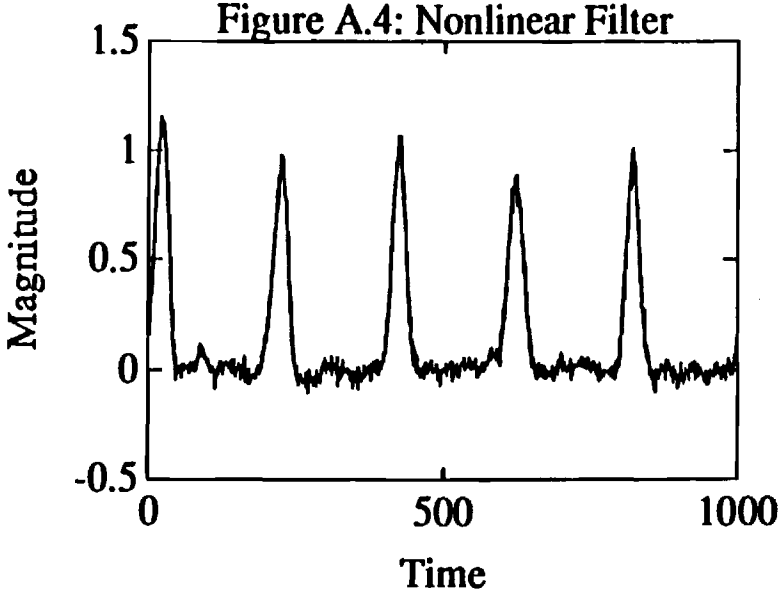
Figure A.4: Nonlinear Filter
Magnitude
1.5
1
0.5
0
-0.5
0
500
1000
Time

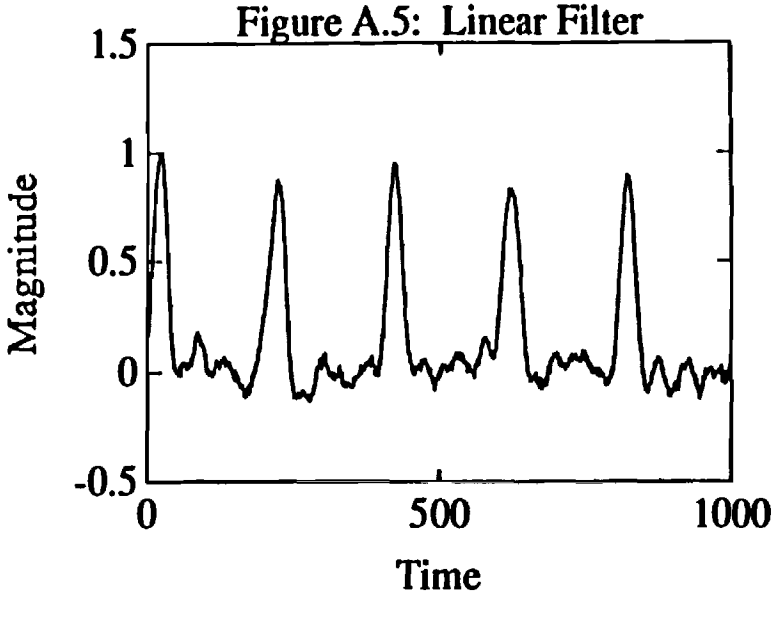
Figure A.5: Linear Filter
Magnitude
1.5
1
0.5
0
-0.5
0
500
1000
Time

Appendix B: Recursive Wiener Filter Example

In this appendix, it is shown how a time varying neural network was trained using backpropagation through time to emulate a Recursive Wiener filter. It is shown that the signal to noise ratio can be significantly increased using this technique if the signal is correlated over time.

The input signal of the example we shall consider is corrupt with gaussian white noise and is correlated with the signal at the previous time step. The equation for the signal without noise added, which shall be referred to as the desired signal, at each time interval is given by the following equation.

$$d(n+1) = d(n) + 1$$

The desired signal at time zero is gaussian with mean 0, variance 10. The observed signal, which is simply the desired signal with noise added, is input to the filter at each time step. The observed signal is determined by the following equation.

$$x(n) = d(n) + r(n)$$

where r(n) is a gaussian random variable with mean 0, variance 10. Figures B.1 and B.2 show an example of d(n) and x(n) propagated in time.

A time varying neural network was trained using x(n) and the output of the filter at the previous time step as the inputs and d(n) as the desired responses. After the network was trained, the signal shown in Figure B.2 was used as input to the filter. The resulting output of the filter is shown in Figure B.3. It can be observed that the time varying neural network was able to significantly reduce the noise present at the output.

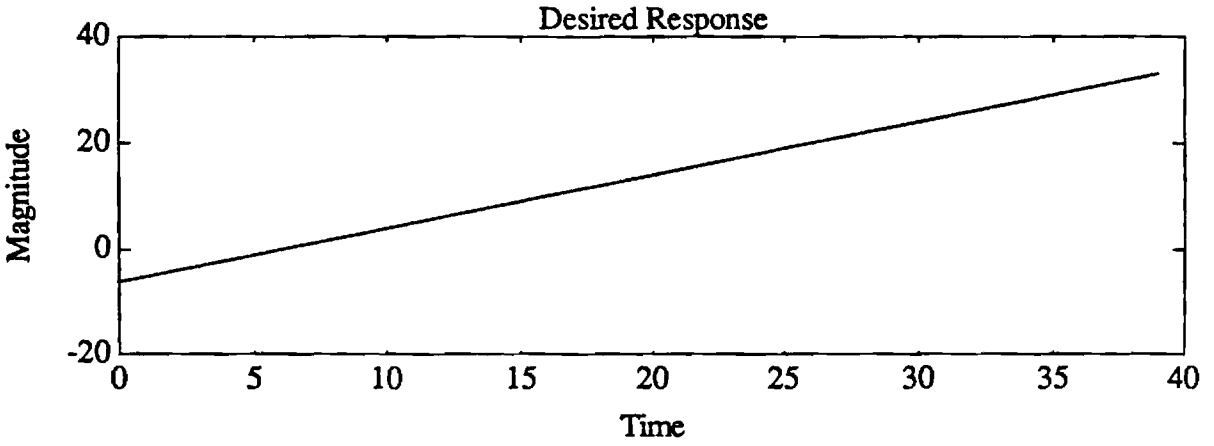


Figure B.1: Desired Response

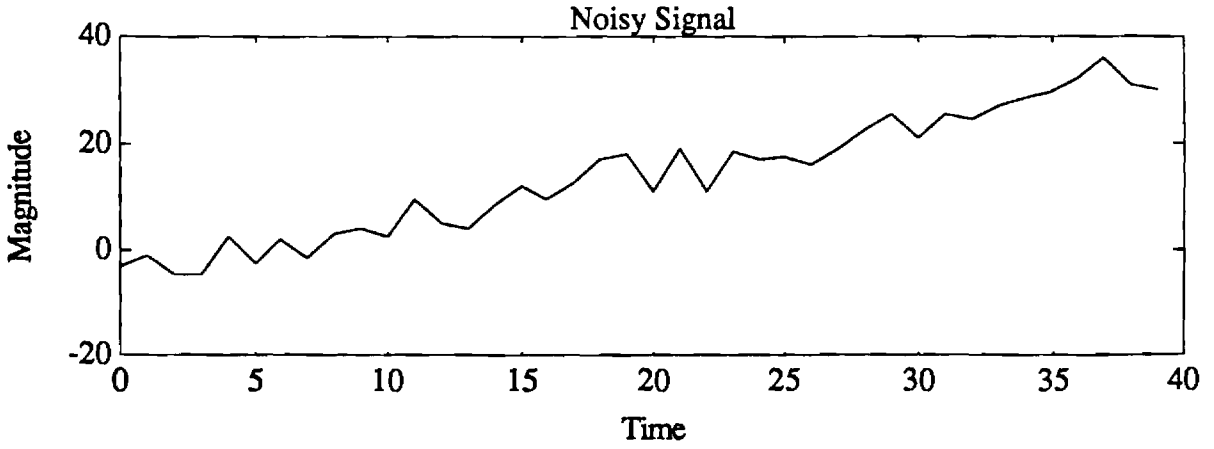


Figure B.2: Noisy Signal

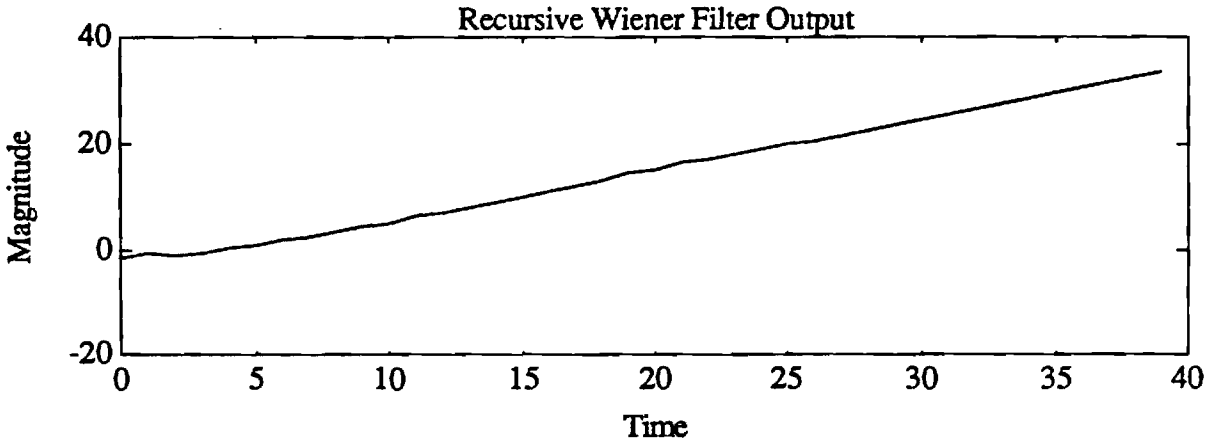


Figure B.3: Recursive Wiener Filter Output

References

1. Widrow, B. and Stearns, S. (1985). Adaptive Signal Processing. New Jersey: Prentice-Hall.

2. Pearlmutter, B. (1989). Learning State Space Trajectories in Recurrent Networks. A paper presented at the International Joint Conference on Neural Networks, Washington D.C.

3. Gray, R. (1984). Vector Quantization. IEEE ASSP Magazine, April, pp. 4-29.

# Neural Networks Communication Processor

Second Quarterly Report
(August 1989 - October 1989)

Prepared by

B. Widrow, S. Piche

Information Systems Laboratory
Department of Electrical Engineering
Stanford University, Stanford, CA 94305

for

Georgia Institute of Technology
Subcontract No. E-21-T22-S1

and

Rome Air Development Center
Contract No. F30602-88-D-0025

# Table of Contents

## 1.0 Introduction

The first quarterly report identified several problems in the field of communications where neural networks might offer solutions. This report presents neural network architectures and algorithms which may offer solutions for some of the problems presented in the first report.

Architectures and algorithms for noise reduction and vector quantization are discussed in the first two sections. The architectures for noise reduction are feedforward sigmoidal neural networks and feedback sigmoidal neural networks. Backpropagation is used to train the feedforward network while a novel algorithm called the on-line recursive algorithm is use to train the feedback network. The structure used for the vector quantizer is a two layer one hidden node neural network. A modified backpropation algorithm is used to train the network. A brief discussion of work related to multilayer linear network convergence and adaptive jammer direction identification is included in the last section.

## 2.0 Noise Reduction

In the first quarterly report, two types of possible neural network noise reduction systems were discussed: nonlinear noise reduction systems and time varying noise reduction systems. This section consists of an analysis of neural network architectures for these types of noise reduction systems.

### 2.1 Nonlinear Noise Reduction Architecture

As was discussed in the first quarterly report, the nonlinear noise reduction filter is required when the noise added to the signal is passed through a nonlinearity (Figure 1). Since the noise introduced to the signal is a nonlinear function of the reference noise, a nonlinear adaptive filter is required. Both feedforward and feedback neural networks have been suggested for the nonlinear adaptive filter.

The feedforward neural network architecture chosen for the adaptive filter is a multiple layer network with sigmoids located at outputs of all hidden nodes. The inputs and output of the feedward neural network are exactly the same as the inputs and outputs of a finite impulse response linear filter. This architecture was chosen because it can implement nonlinear functions and has a well know training algorithm associated with it, backpropagation. The implementation of the feedforward neural network into the filter shown in Figure 1 is not difficult. The error used in the backpropagation algorithm is the output of the filter.

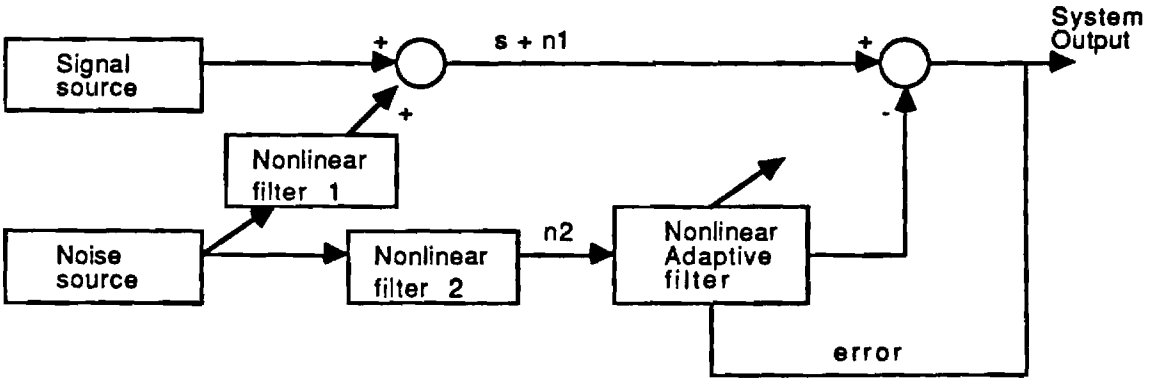


Figure 1: Nonlinear Adaptive Noise-Cancelling Concept

The feedback neural network structure is similar to the feedforward structure except that delayed versions of the output are feedback to the input of the neural network as shown in Figure 2. The inputs and output of the feedback neural network are exactly the same as that of an infinite impulse response filter. Feedback is introduce into the adaptive filter in order to reduce the number of inputs and size of the neural network.

In order to adapt the feedback architecture, it was necessary to device an on-line algorithm. The standard method for adapting feedback neural networks is to use backpropagation through time. This algorithm requires a termination point in order to calculate the weight changes of the network. This technique is not satisfactory for instances where no termination point exists as is the case in adaptive filtering. In order to alleviate this problem, an on-line recursive algorithm was devised.

## 2.2 The On-Line Recursive Algorithm

The on-line algorithm is used to update the weights of the neural network shown in Figure 2. The input of the neural network is composed of delayed input signals and delayed output signals which are represented by the following vectors.

$$\mathbf{X} = \begin{bmatrix} x_{k-n} \\ x_{k-(n-1)} \\ \cdot \\ \cdot \\ \cdot \\ x_k \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} y_{k-n} \\ y_{k-(n-1)} \\ \cdot \\ \cdot \\ \cdot \\ y_{k-1} \end{bmatrix}$$

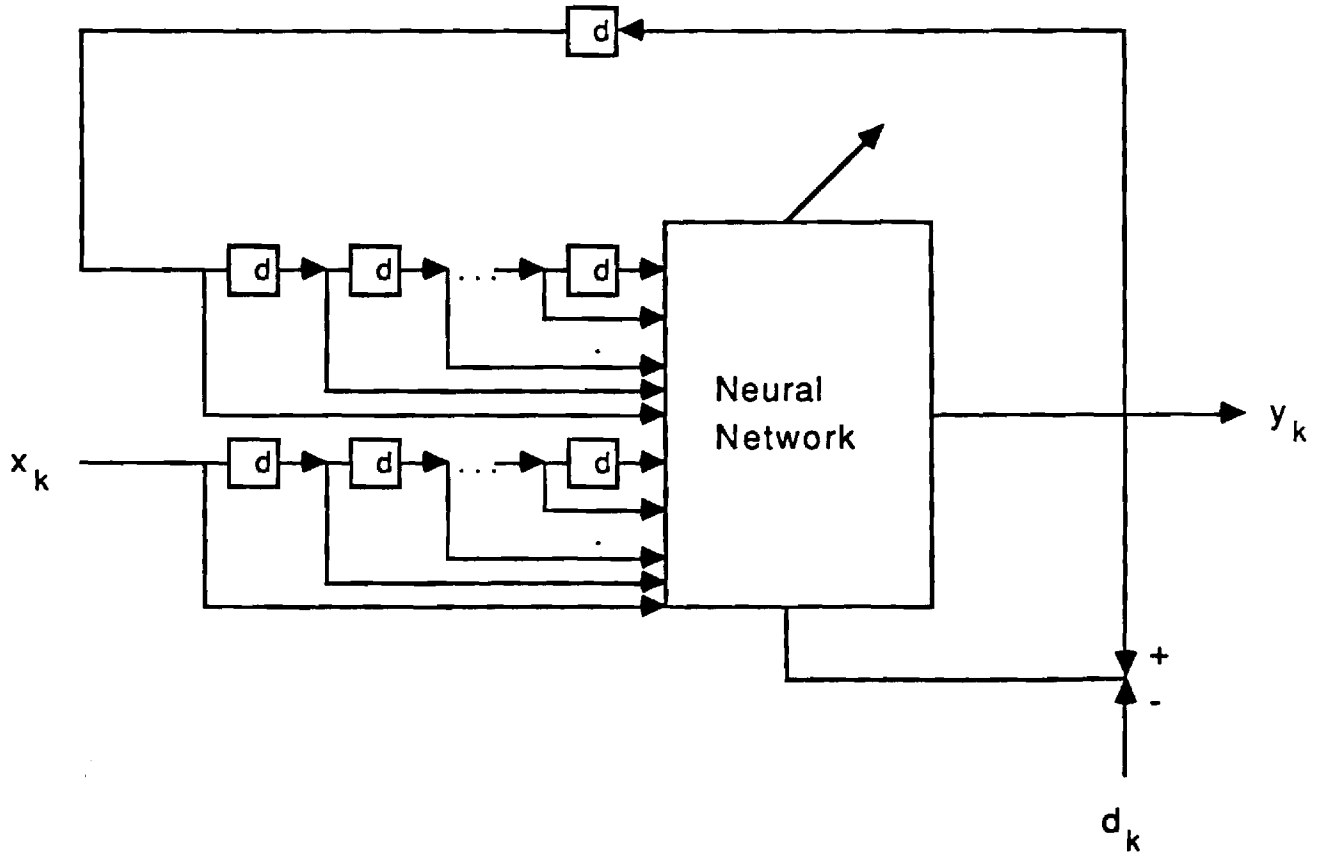


Figure 2: Adaptive Neural Filter

The input vector of the neural network, **Z**, is a combination of the **X** vector and **Y** vector.

$$\mathbf{Z} = \begin{bmatrix} \mathbf{X} \\ \mathbf{Y} \end{bmatrix}$$

The on-line algorithm is used to minimize an arbitrary energy function. In this case, the energy function is chosen to be the square of the difference between the desired response, $d_k$, and the output signal.

$$E_k = \frac{1}{2}(d_k - y_k)^2 \qquad \text{(Eq. 1)}$$

Since the algorithm uses gradient descent to minimize the energy function at each time step, it is necessary to compute $\frac{\partial E_k}{\partial w}$.

$$\frac{\partial E_k}{\partial w} = -(d_k - y_k)\frac{\partial y_k}{\partial w} \qquad \text{(Eq. 2)}$$

$\frac{\partial y_k}{\partial w}$ can be calculated recursively using the following scheme. $\frac{\partial y_k}{\partial w}$ can be decomposed into the summation of two terms. The first term represents the change in $y_k$ due to a change in w at time k. It can easily be calculated using backpropagation and notationally shall be represented by $\frac{\partial y_k}{\partial w_k}$. The second term represents the change in $y_k$ due to a change in w at all time earlier than time k. Using the chain rule, this term can be further be subdivided into two multiplicative parts. The first part is the Jacobian, the derivative of the output with respect to the input vector. The second part is the derivative of the inputs with respect to the weights. Putting together all of the terms discussed above results in the following equation for $\frac{\partial y_k}{\partial w}$.

$$\frac{\partial y_k}{\partial w} = \frac{\partial y_k}{\partial w_k} + \mathbf{J}^T \frac{\partial Z}{\partial w} \qquad \text{(Eq. 3)}$$

where

$$\mathbf{J} = \begin{bmatrix} \frac{\partial y_k}{\partial x_{k-n}} \\ \cdot \\ \frac{\partial y_k}{\partial x_{k-1}} \\ \frac{\partial y_k}{\partial y_{k-n}} \\ \cdot \\ \frac{\partial y_k}{\partial y_{k-1}} \end{bmatrix} \qquad \frac{\partial \mathbf{Z}}{\partial w} = \begin{bmatrix} \frac{\partial \mathbf{X}}{\partial w} \\ \frac{\partial \mathbf{Y}}{\partial w} \end{bmatrix}$$

It should be noted that the equation above is recursive. $\frac{\partial \mathbf{Z}}{\partial w}$ is initially set to zero. ($\frac{\partial \mathbf{X}}{\partial w}$ will always equal zero since the input signal is not dependent on the weights.)

Equations 2 and 3 may be used on line to update the weights. However, as the weights are changed, the accuracy of $\frac{\partial y_k}{\partial w}$ would diminish. In order to prevent this from happening, the second term of equation 4 is multiplied by a forgetting vector. The components of the 2n-1 dimensional forgetting vector, **F**, take on values between 0 and 1.

$$\frac{\partial y_k}{\partial w} = \frac{\partial y_k}{\partial w_k} + \mathbf{J}^T \frac{\partial \mathbf{Z}}{\partial w} \mathbf{F} \qquad \text{(Eq. 4)}$$

All terms of equation 4 are relatively easy to calculate. $\frac{\partial y_k}{\partial w}$ and **J** can be calculated using backpropagation. $\frac{\partial \mathbf{Z}}{\partial w}$ is stored in memory and **F** is set at the beginning of the calculation.

## 2.3 Nonlinear Noise Reduction Results

Both the feedforward and feedback neural networks have been used as the adaptive filter of the nonlinear noise reduction filter in several test cases. In all of these cases, the nonlinear noise reduction filter was able to perform noticeably better than the linear noise

reduction filter. Although results have been achieved which validate the theory of such filters, further work is need to explore their properties.

## 2.4 Time Varying Neural Networks for Noise Reduction

Investigations of using backpropagation through time to implement time varying neural networks has continued. Currently, an implementation of a time varying neural networks is being coded for state estimation of a dynamic system. (The dynamic system chosen is a cart-pole balancing system.) This example is being coded to test the practicality of using a time varying neural network for noise reduction.

# 3.0 Vector Quantization

In the first quarterly report, vector quantization was discussed as a possible application for neural networks in communication. In particular, tree structured vector quantizers were suggested. In this report, a neural network architecture and algorithm is presented.

## 3.1 Node Architecture

As its name indicates, a tree structured vector quantizer is implemented using a tree structure. The tree structure consists of both nodes and branches. Each node represents a binary decision while each branch represents the control flow from an upper node to a lower node based on the binary decision of the upper node. As an example, a tree structured vector quantizer implemented with three data bits would consist of one top node, two second layer nodes and four third layer nodes as shown in Figure 3. The first data bit would be produced by the top layer, the second data bit would be produced by the second layer and the third data bit would be produced by the third layer.

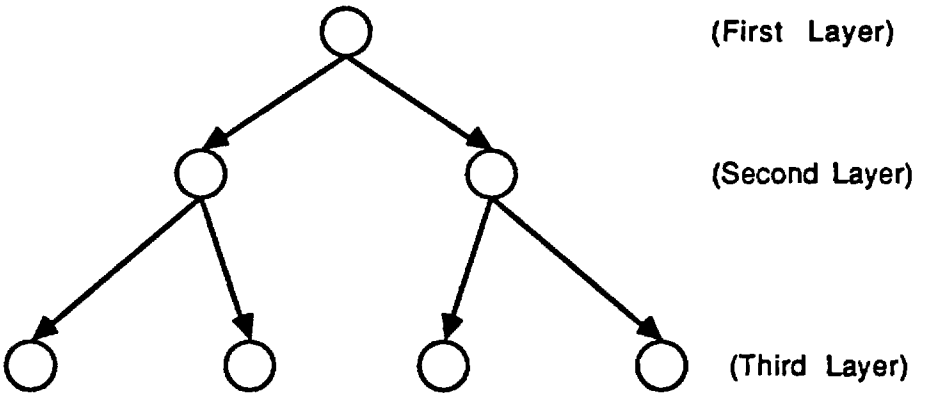


Figure 3: Tree Structure

The neural network architecture proposed to implement a tree structured vector quantizer would use the structure shown in Figure 4 for each of the nodes in the tree structure. The left side of the network performs a one bit encoding of the input vector. The output of the left side is not only used to produce a bit value, it is also used to direct the flow to a node in the next layer. The right side of the structure is used to decode the data bit. The quantized vector is produced by the decoder output of a final layer node which is selected using the data bits.

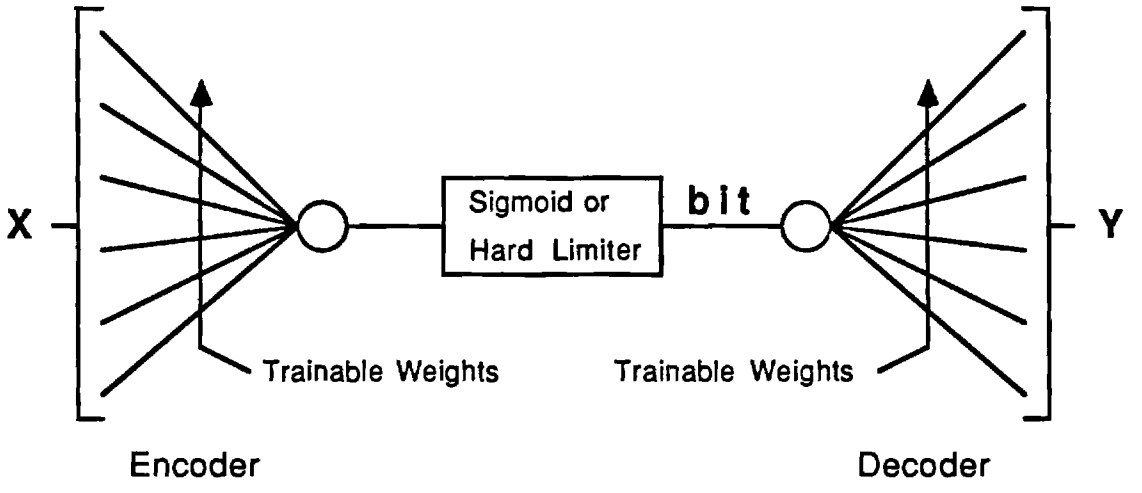


Figure 4: Vector Quantizer Node Structure

## 3.2 Training Algorithm

Each of the nodes of the vector quantizer must be trained. Training proceeds in the following manner:

1. Choose a vector from the training set.

2. Forward propagate the vector through the network representing the appropriate node.

3. Using the input vector as the desired output vector, backpropagate the error gradient through the network. Based on the gradients, update the weights.

4. Using the binary decision of the network, select the next appropriate node. Repeat steps 2 and 3. Continue in this manner until reaching the bottom of the tree.

5. Cycle through the training set using steps 1-4 until the weights of all the nodes stabilize. Once stabilization occurs, the sigmoid of each node is replace with a hard limiter. Once again, cycle through the training set until the weights stabilize.

## 3.3 Results

Promising preliminary results have been obtained using small tree structures. Future work shall include the training of larger tree structures and possible embellishments of the network structure and algorithm.

# 4.0 Additional Areas of Research

The first quarterly report briefly discussed two additional subjects, faster linear convergence and jammer direction identification. Upon further examination of using multilayered linear networks to provide faster convergence, it was concluded that in general multilayered linear networks would probably not decrease convergence time. Therefore, investigation of this subject has been discontinued.

Using neural networks for jammer direction identification is still under investigation. However, due to the amount of time invested in noise reduction and vector quantization, no significant results have been obtained.

## 5.0 Conclusion

Neural network architectures and algorithms for implementation of both noise reduction systems and vector quantizers were presented in this report. Preliminary results indicate that these architectures and algorithms work as predicted by theory. Future effort shall concentrate on further investigation of the work outlined in this report.

# Neural Networks Communication Processor

Final Report

Prepared by

B. Widrow, S. Piche

Information Systems Laboratory
Department of Electrical Engineering
Stanford University, Stanford, CA 94305

for

Georgia Institute of Technology
Subcontract No. E-21-T22-S1

and

Rome Air Development Center
Contract No. F30602-88-D-0025

## 1.0 Final Report Package

The final report package contains three documents. The first, which is this document, summarizes three important conclusions which were reached while working on this contract. In addition, a brief discussion of future work related to the results of our current work is included. The second document is a paper by Stephen Piche titled "First-order gradient descent for discrete-time dynamic networks." A version of this paper will be submitted to IEEE Transactions on Neural Networks. The bulk of the work on the last phase of this contract concentrated upon putting together the paper by Stephen Piche. We submit this paper as the required final technical report. The third document is a paper titled "Temporal backpropagation for FIR neural networks" by Eric Wan, who was partially supported by this contract. This paper appeared in the June, 1990, International Joint Conference on Neural Networks Proceedings and will also appear in the Proceedings of the 1990 Connectionists Models Summer School.

## 2.0 Brief Conclusions

Based upon the research of the past year, we have come to three important conclusions which are briefly stated below. For more detail on conclusions 1 and 2, a reading of the papers by Piche and Wan is suggested.

1. Adaptive Nonlinear IIR Filtering

Currently, communication processing is highly dependent upon adaptive filtering technology. Neural networks are important to the field of communication processing because they allow the possibility of nonlinear adaptive filtering. In our lab, two techniques for adapting IIR nonlinear filters have recently been developed. The development of these algorithms opens up a whole new area of research in the field of communication processing. Like the linear IIR filter, the neural network IIR filter should offer higher performance for many practical applications. The paper by Piche discusses in detail two different types of algorithms which can be used for adapting nonlinear IIR filters based upon neural networks. The two algorithms are presented in their most general form. This allows the algorithms not only to be used for adaptive filtering but also for adaptive controller design.

2. Adaptive Nonlinear FIR Filtering

The use of neural networks for nonlinear FIR filtering is well established. Generally, by unraveling the dynamic network to form a static network, backpropagation can be used to train the network. However, in the paper by Wan, it is shown that there exists a computationally more efficient method of computing the gradient than using the backpropagation algorithm. Because the computational requirements of any algorithm limits its applicability to problems, the FIR technique introduced by Wan should allow greater use of nonlinear FIR filters.

3. Nonlinear Noise Cancelling System

A noise cancelling system is used to eliminate noise from a corrupted signal assuming that a noise reference signal is available. In general, the noise added to the signal to form the corrupted signal is a filtered version of the noise reference signal. A noise cancelling system works by training an adaptive filter to mimic the noise filter. By subtracting the output of the adaptive filter from the corrupted signal, the original signal is recovered. We concluded that it is possible to use the adaptive noise cancelling system even when the noise filter is nonlinear. In this case, a nonlinear adaptive filter, based upon a neural network, can be trained to mimic the nonlinear noise filter. The example section in the paper by Piche includes a nonlinear noise cancelling system example. The filter in this example is not only nonlinear but also IIR. We were particularly interested in the adaptive noise cancelling system because of its close relationship to adaptive beamformers.

## 3.0 Future Work - Adaptive Beamformers

Future work should concentrate upon application of adaptive nonlinear filters. Much of the time spent on this contract during the past year went into developing the algorithms for adapting the nonlinear filters leaving little time for extensive testing of the algorithms with practical applications. Besides practical applications, the stability and convergence rate of the IIR algorithms needs further investigation. In addition, a detailed comparison of the differences between the FIR and IIR algorithms is needed.

The applicability of the algorithms to the adaptive noise cancelling system gives a clear indication that neural networks can be used for adaptive beamforming. Two important questions related to adaptive beamforming with neural networks remain unanswered and need to be

addressed in the future. Does a nonlinear adaptive filter (neural network) offer any advantage in the adaptive beamforming problem and could a linear neural network be used instead of a linear filter to speed convergence of a gradient descent algorithm?

# FIRST-ORDER GRADIENT DESCENT TRAINING of ADAPTIVE DISCRETE-TIME DYNAMIC NETWORKS

Four basic algorithms for adapting dynamic neural network filters, which can be used for communication processing, are derived and analyzed.

Stephen W. Piché and Bernard Widrow
Department of Electrical Engineering, Stanford University
Stanford, CA 94305

January 3, 1991

Final Report

Prepared for
Georgia Institute of Technology
School of Electrical Engineering
Atlanta, GA 30332-0250

Rome Air Development Center
Griffiss Air Force Base, NY 13441-5700

# *Abstract*

This paper describes the training of discrete-time dynamic systems with adaptive parameters (recurrent neural networks) using first-order gradient descent algorithms. To facilitate the explanation of these algorithms, a standard representation of a discrete-time dynamic system is defined. Any differentiable discrete dynamic system may be put in this standard representation and trained using a gradient descent algorithm. Using the standard representation, we describe two general types of learning algorithms. The first is based upon the discrete-time Euler-Lagrange equations, and the second is based upon a recursive update of the output gradients. Both the epochwise and on-line versions of these algorithms are presented. When the dynamic system is implemented by a neural network, the epochwise algorithm based on the Euler-Lagrange equations is equivalent to backpropagation-through-time and the on-line method based on the recursive equation is the same as recursive backpropagation. It is shown that the epochwise versions of the algorithms are equivalent. The two on-line versions of the algorithms are shown to be approximately equivalent. Because of the equivalence of the algorithms, selection of an appropriate gradient descent algorithm is based solely upon computational efficiency and storage requirements. Accordingly, a discussion of these two properties of the algorithms is included. To illustrate the differences between the algorithms and the usefulness of the standard representation, two examples are included.

# 1 Introduction

The ability of humans to control and interact with a complex environment has motivated our study of adaptive discrete-time dynamic systems which are fully or at least partially composed of neural networks. Humans regularly perform certain tasks easily and proficiently which have proven difficult to reproduce with machines. Examples of such tasks include driving a car, recognizing the differences between a cat and a dog, and understanding spoken language. Humans learn how to accomplish these tasks in part by interacting with, manipulating and eventually controlling their environment. In order to build machines which accomplish these difficult tasks, it may be necessary both to mimic humans learning through environmental interaction and to model the low level functions of the human nervous system. The algorithms presented in this paper provide one possible technique for accommodating both of these requirements. These algorithms train neural networks, which model the low level functions of the human nervous system, by interacting with a user-specified environment.

An adaptive dynamic system which is composed of a neural network and a set of equations which describe the environment may be used as an adaptive model of the interaction of a human with its environment. In this model, the neural network receives the state of the environmental as input. Using the state in combination with a desired goal, the neural network outputs a control signal which manipulates the environment. Using the training algorithms presented in this paper, system performance often becomes strongly human-like. The first example of Section 9 gives a prime example of this phenomenon.

Although the human ability to control its environment motivates us, our primary interest lies in the development of engineering tools rather than in the modeling of humans. The training of discrete-time dynamic systems has applications in the engineering fields of pattern recognition, nonlinear control, adaptive control and adaptive digital filtering. It is in these areas that the material in this paper will be of most immediate use.

Currently, the theory on first-order gradient descent training of adaptive discrete-time dynamic networks is described in separate and unrelated terms in several papers by different authors [1,2,3,4,5,6]. Our goal is both to bring together in a coherent manner and to expand the theory on this subject. A coherent presentation of the subject is achieved by deriving the learning algorithms using a standard representation of a dynamic system. The derivation of the generalized forms of the existing algorithms provides for extensions of current algorithms. Bringing together and generalizing the theory in this manner should facilitate the selection of appropriate gradient descent algorithms for problems requiring discrete-time recurrent networks. It should be noted that a forthcoming paper by Williams and Zipser [7] contains a detailed discussion of adapting dynamic neural networks, whereas, this paper presents the concepts of first-order gradient descent for systems composed both fully or partially of neural networks.

The paper is composed of ten sections. Section 2 introduces notation and the standard representation for a discrete-time dynamic system. Section 3 presents the differences between on-line and epochwise training. An introduction to gradient descent training of static systems is given in Section 4. Section 5 presents the algorithms used for adapting discrete-time dynamic systems. The equivalence of the algorithms is presented in Section 6. A comparison of the computational and storage requirement of the algorithms is included in Section 7. Two techniques of speeding-up the on-line training algorithms are discussed in Section 8. Section 9 presents two applications which illustrate the usefulness of the theory discussed in the paper, and Section 10 provides a conclusion.

# 2 System Definition

In this section, a standard representation of any discrete-time dynamic system is proposed. This representation is used in the derivation of the learning algorithms. In addition, the ordered derivative, which simplifies the calculation of derivatives of complex systems, is introduced.

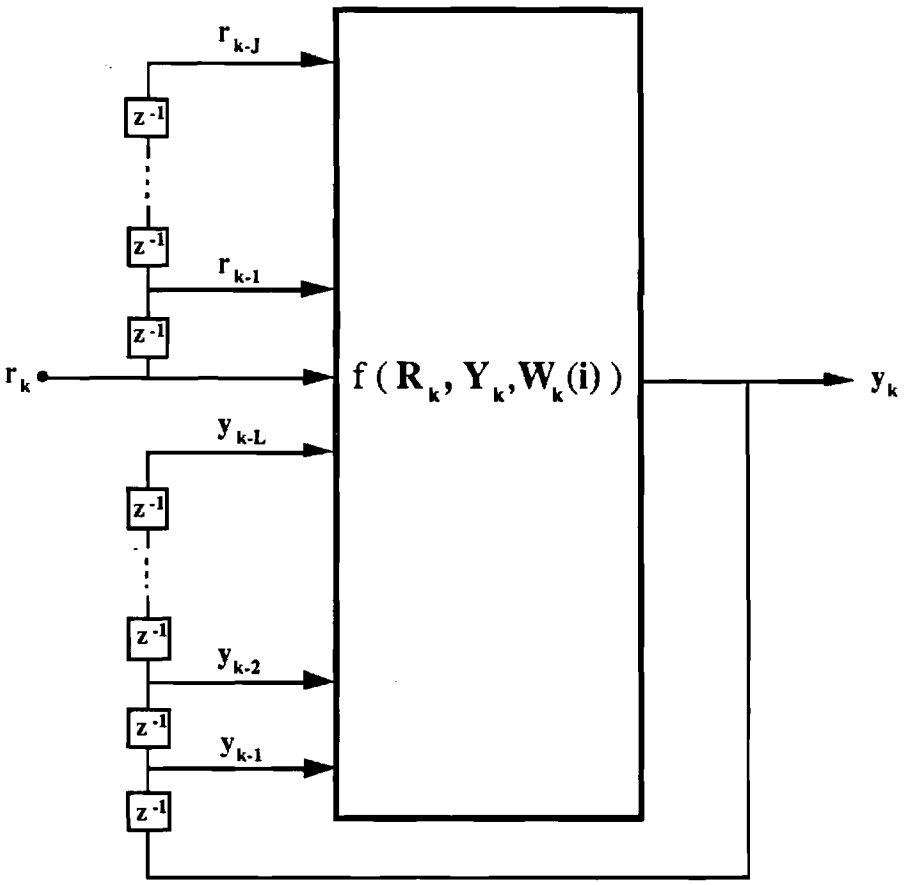


Figure 1: Standard Representation.

## 2.1 The Standard Representation

Let $k$ denote the iteration of a discrete-time dynamic system, with $k = 0$ representing the first iteration of the system. A standard representation of discrete-time dynamic system is illustrated in Figure 1. The input of the system at iteration $k$ is defined by two components. The first component, $\mathbf{R}_k$, is composed of an external input vector, $\mathbf{r}_k$, and the previous $J$ delayed versions of this vector, $\mathbf{r}_{k-1}, \ldots, \mathbf{r}_{k-J}$. The external input vector is a length $M$ column vector, $\mathbf{r}_k \in R^{[M \times 1]}$. Therefore, the external inputs to the system including delayed inputs is a length $(J+1)M$ column vector of the form $\mathbf{R}_k = [\mathbf{r}_k^T, \mathbf{r}_{k-1}^T, \ldots, \mathbf{r}_{k-J}^T]^T \in R^{[(J+1)M \times 1]}$. The second component, $\mathbf{Y}_k$, is made up of the previous $L$ output vectors. The output vector at iteration $k$ is a length $N$ column vector, $\mathbf{y}_k \in R^{[N \times 1]}$. Therefore, $\mathbf{Y}_k$ is a length $LN$ column vector of the form $\mathbf{Y}_k = [\mathbf{y}_{k-1}^T, \mathbf{y}_{k-2}^T, \ldots, \mathbf{Y}_{k-L}^T]^T$.

At iteration $k$, let the adaptive parameter vector, which we shall refer to as the weight vector, be selected from a set of weight vectors. In general, this set of weight vectors is generated by the training algorithm as discussed in Section 3. Assuming a weight vector to be a length $Q$ column vector, the weight set $\mathbf{W}$ has the form $\mathbf{W} = \{\mathbf{W}(0), \mathbf{W}(1), \ldots, \mathbf{W}(i), \ldots\}$. The use of the $i^{th}$ weight vector at the $k^{th}$ iteration shall be denoted $\mathbf{W}_k(i)$. Finally, let $w_k(i)$ denote any weight of the vector $\mathbf{W}_k(i)$. Of course, $w_k(i)$ is a scalar.

By denoting any element of a discrete-time dynamic system which is connected to a delay as an output and including it in $\mathbf{y}_k$, any discrete-time system can be written as

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)) \tag{1}$$

where the function $f$ contains no delay. Thus the schematic representation in Figure 1 can be used to

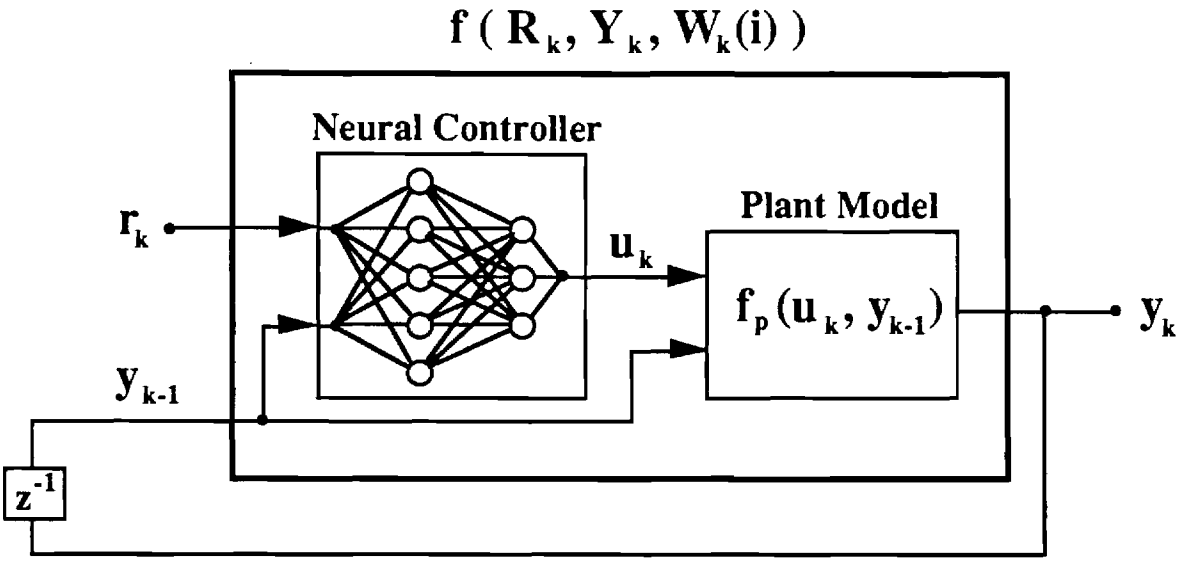


Figure 2: Neural Controller and Plant in the Standard Representation.

describe any discrete-time dynamic system. The first-order gradient descent algorithms to be described in this paper are defined in terms of the standard representation shown in Figure 1. Because any adaptive discrete-time dynamic system can be put in the standard representation, such systems can be trained using the algorithms described in this paper provided that the output vector, $\mathbf{y}_k$, is differentiable with respect to the weight vector, $\mathbf{W}_k(i)$, and the recurrent input, $\mathbf{Y}_k$. The need for this requirement will become evident in Section 5.

## 2.2 An Example of a System in the Standard Representation

In order to illustrate the use of the standard representation, an adaptive discrete-time dynamic system which consists of a neural network and an environment model, is shown schematically in Figure 2. In this figure, we use the traditional term *plant model* instead of *environmental model*. We define the dynamic system of Figure 2 to be a neural network controller-plant system.

Because we often use the controller-plant system to illustrate important points throughout this paper, it is useful at this point to discuss this system in greater detail. The plant model may take two different forms. The most general is simply a set of equations which map previous states and current control to the next state, $\mathbf{y}_k = f_p(\mathbf{u}_k, \mathbf{Y}_k)$, where $\mathbf{Y}_k$ are the previous states of the plant and $\mathbf{u}_k$ is the control signal vector. If the equations of the model are nonlinear, adapting the structure of Figure 2 results in a neural controller which implements a nonlinear state feedback control law. This technique provides a method of designing a nonlinear controller for a nonlinear plant.

Using the second form of a plant model, a neural network model, has applications in the field of adaptive control [3,5,8]. In this case, the plant model takes the form $\mathbf{y}_k = f_p(\mathbf{u}_k, \mathbf{Y}_k, \mathbf{W}^p)$, where $\mathbf{W}^p$ is the weight vector of the neural network model. This model can be updated on-line using plant input-output data. Because the algorithms presented in this paper use the plant model to update the controller, the neural network controller can adapt to changes in the plant.

## 2.3 Ordered Partial Derivatives

Because we are interested in using first-order gradient descent to update the adaptive weights, we are required to calculate a partial derivative of the associated dynamic system. The ordered partial derivative, which is a special partial derivative for an ordered set of equations, provides a mathematical technique for easily

finding derivatives of complex dynamic systems [6].

In order to discuss the ordered derivative, we must first introduce the concept of an ordered set of equations. Let $\{z_1, \ldots, z_i, \ldots, z_j, \ldots, z_n\}$ be a set of $n$ variables whose values are determined by a set of $n$ equations. This set of equations is defined to be an ordered set of equations if each variable $z_i$ is a function only of the variables $\{z_1, \ldots, z_{i-1}\}$. Thus, the equation for any variable of an ordered set of equations can be written as

$$z_i = f(z_1, \ldots, z_{i-1})$$

Because of the ordered nature of this set of equations, the variables $\{z_1, \ldots, z_{i-1}\}$ must be calculated before $z_i$ can be computed.

When calculating the partial derivative of a variable it is necessary to specify which variables are held constant and which are allowed to vary. Typically, if this is not specified, it is assumed that all variables are held constant except those terms appearing in the denominator of the partial derivative. This is the convention we have adopted in this paper.

The ordered partial derivative, which is defined only for variables of an ordered set of equation, is a partial derivative whose constant and varying terms are determined using the ordered set of equations. The constant terms of the order partial derivative of $z_j$ with respect to $z_i$, which is denoted $\partial^+ z_j / \partial z_i$ in order to distinguish it from an ordinary partial derivative, are $\{z_1, \ldots, z_{i-1}\}$. The varying terms are $\{z_i, \ldots, z_j, \ldots, z_n\}$.

The ordered derivative is usually found using either of two chain rule expansions. The first expansion, which is expressed in vector form as

$$\frac{\partial^+ \mathbf{z}_j}{\partial \mathbf{z}_i} = \frac{\partial \mathbf{z}_j}{\partial \mathbf{z}_i} + \sum_{k=i}^{j} \frac{\partial^+ \mathbf{z}_j}{\partial \mathbf{z}_k} \frac{\partial \mathbf{z}_k}{\partial \mathbf{z}_i} \tag{2}$$

was shown by Werbos in his thesis [6]. The proof of the second chain rule expansion

$$\frac{\partial^+ \mathbf{z}_j}{\partial \mathbf{z}_i} = \frac{\partial \mathbf{z}_j}{\partial \mathbf{z}_i} + \sum_{k=i}^{j} \frac{\partial \mathbf{z}_j}{\partial \mathbf{z}_k} \frac{\partial^+ \mathbf{z}_k}{\partial \mathbf{z}_i} \tag{3}$$

uses arguments similar to those used to prove the first expansion.

Finally, one comment on mathematical notation, throughout this paper it is assumed that a partial derivative of the form, $\partial \mathbf{a} / \partial \mathbf{b}$, where $\mathbf{a} \in R^{[A \times 1]}$ and $\mathbf{b} \in R^{[B \times 1]}$, is a matrix of the form $R^{[A \times B]}$.

# 3 Epochwise and On-Line Training

Any adaptive algorithm adjusts the parameters of a system so that the system responds to a set of inputs in some desired manner. First-order gradient descent algorithms accomplish this goal by minimizing an error function. The definition of this error function is dependent upon whether the system is operating in an epochwise or on-line mode. In this section, both epochwise and on-line training are defined. The epochwise and on-line error functions as well as their associated weight update equations are also presented.

## 3.1 Epochwise Training Algorithms

An epoch is a forward iteration of the dynamic system from iteration $k = 0$ to $k_f$, where $k_f$ is the final iteration. An epochwise training algorithm is any algorithm in which training takes place after each epoch or a series of epochs of the dynamic system.

In order to use the gradient descent epochwise algorithms, an error must be defined. It is common for this error to be a function only of the outputs of the dynamic system, $\{\mathbf{y}_0, \ldots, \mathbf{y}_{k_f}\}$, and a set of desired

output vectors, $\{\mathbf{d}_0, \ldots, \mathbf{d}_{k_f}\}$, with $\mathbf{d}_k \in R^{[N\times 1]}$. The desired output vectors and the set of inputs vectors associated with the desired output vectors are given in a training set. This set is composed of $P$ elements with element $p$ taking the form $\{\mathbf{R}_{0p}, \mathbf{Y}_{0p}, \mathbf{r}_{1p}, \ldots, \mathbf{r}_{k_f p}, \mathbf{d}_{1p}, \ldots, \mathbf{d}_{k_f p}\}$. It should be noted that the desired response need not be defined for each iteration, only for the final iteration $k_f$. A commonly used epochwise error function is

$$E = \sum_{p=0}^{P} \sum_{k=0}^{k_{fp}} \frac{1}{2}(\mathbf{d}_{kp} - \mathbf{y}_{kp})^T(\mathbf{d}_{kp} - \mathbf{y}_{kp}) \tag{4}$$

which is the sum of the squared error over the entire training set. The error of Equation 4 is calculated using an ordered set of equations. Because of the ordering of this set, the error is always the last calculation performed in this set.

Utilizing gradient descent, the epochwise algorithms presented in this paper update the weights using

$$w(i+1) = w(i) - \mu \frac{\partial^+ E}{\partial w(i)} \tag{5}$$

where $\mu$, the learning rate, is a suitably chosen positive constant. The update rule generates a new weight vector $\mathbf{W}(i+1)$ from the vector $\mathbf{W}(i)$. If the error function defined in Equation 4 is used, the weights are updated after cycling through the training set. Therefore, if weight vector $\mathbf{W}(1)$ is used for the first cycle through the training set, weight vector $\mathbf{W}(i)$ is used for the $i^{th}$ cycle through the set. Generally, the weight vector of the first cycle, $\mathbf{W}(1)$, is randomly initialized.

Although first-order gradient descent provides the basis for adaptation of the weights, the algorithms discussed in this paper have come to be known by the method for which they calculate the error gradient of Equation 5. Hence, it should be remembered, that even though backpropagation-through-time and recursive backpropagation, two algorithms presented in Section 5, have quite different names, they both perform first-order gradient descent.

## 3.2 On-line Training Algorithms

If the weight update of an algorithm at the current iteration $k'$ depends only on the states of the system at iterations $\{k', k'-1, k'-2, \ldots\}$, then the algorithm is defined to be an on-line training algorithm. The implied dependence only upon the current and past values of the system allows the weight updates to be computed in real-time in most cases. The key difference between on-line and epochwise training algorithms is that an on-line algorithm adapts the weights of the system as it runs while an epochwise training algorithm only updates the weights after the final iteration. The primary reason for using an on-line algorithm is that as the number of iterations in an epoch becomes very large, it becomes computationally inefficient to update the weights only after each epoch. Therefore, on-line algorithms, which adapt the weights as the system runs, must be used.

In the on-line case, an error is defined for each iteration. At the current forward iteration, $k'$, the error, $E_{k'}$, is often a function of the desired response vector, $\mathbf{d}_{k'}$, and the output vector, $\mathbf{y}_{k'}$. It is common to use the on-line error function

$$E_{k'} = \frac{1}{2}(\mathbf{d}_{k'} - \mathbf{y}_{k'})^T(\mathbf{d}_{k'} - \mathbf{y}_{k'}). \tag{6}$$

It is well known that in the on-line case, minimization of Equation 6 using first-order gradient descent at each iteration results in the minimization of the mean square error [9]; therefore, using the error defined by Equation 6 minimizes

$$\mathbf{E}[E_{k'}] = \mathbf{E}[\frac{1}{2}(\mathbf{d}_{k'} - \mathbf{y}_{k'})^T(\mathbf{d}_{k'} - \mathbf{y}_{k'})]$$

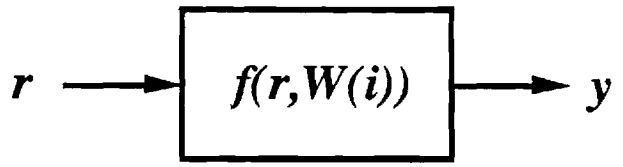


Figure 3: Static System.

where **E** is the expected value operator.

Often, on-line training algorithms update the weights at each iteration based upon the gradient of the error function. At iteration $k'$, the on-line update rule is expressed as

$$w(k'+1) = w(k') - \mu \frac{\partial^+ E_{k'}}{\partial w(k')} \tag{7}$$

where $\mu$ is a suitably chosen constant and $E_{k'}$ is the appropriate on-line error function. Equation 7 is usually initialized by a random setting of $w(0)$. The application of Equation 7 generates a new vector of weights at each iteration. When using an on-line algorithm, it is common for the weights at the iteration $k'$ to be selected from the vector $\mathbf{W}(k')$, therefore, $w_{k'}(k') = w(k')$.

# 4 Static System Algorithms

In order to facilitate the discussion of the training algorithms for discrete-time dynamic systems, it is useful to introduce the first-order gradient descent algorithms for static systems. A static system contains no feedback, therefore, a static system has the structure shown in Figure 3, where $\mathbf{r} \in R^{[M\times 1]}$ is the input vector and $\mathbf{y} \in R^{[N\times 1]}$ is the output vector. A static system can be described by the following equation

$$\mathbf{y} = f(\mathbf{r}, \mathbf{W}(i)).$$

## 4.1 The Backpropagation Algorithm

As in the dynamic system case, the first-order gradient descent techniques for static systems depends upon minimizing an error. In general, this error is a function of the output, and the output is a function of the weights. Therefore, using the chain rule of Equation 2, the error gradient may be expressed as

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial^+ E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w(i)} = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial w(i)}.$$

Assuming that the appropriate equations for the error and the output are available and differentiable, an expression for the error gradient with respect to each weight can be found by differentiating the error and output equations.

If more is known about the structure of the system, it is possible to use this information to decrease the number of computations needed to find the error gradients. The backpropagation algorithm of Rumelhart *et al* [4] does precisely this. Based upon the fact that the static system is composed of a layered feedforward neural network, the backpropagation algorithm efficiently computes the error gradients for such a static network. Using the backpropagation algorithm, the error gradient

$$\frac{\partial^+ E}{\partial w(i)} = \lambda \frac{\partial \mathbf{y}}{\partial w(i)} \tag{8}$$

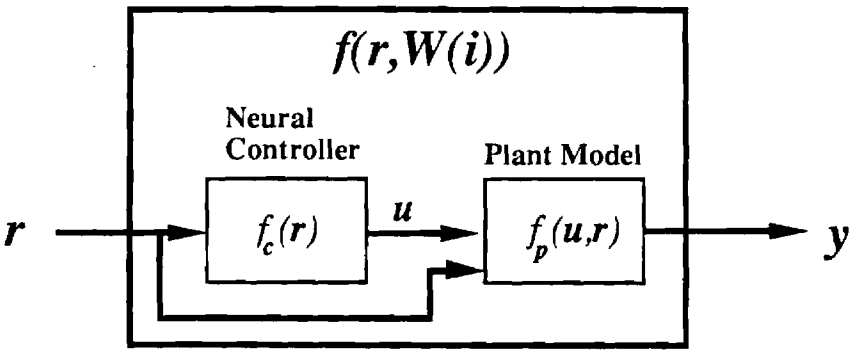


Figure 4: Static Neural Network Control System.

where

$$\lambda = \frac{\partial E}{\partial \mathbf{y}}.$$

is calculated by backpropagating the vector, $\lambda$, through the neural network. It should be noted that any equation which takes the form of Equation 8, can be calculated using backpropagation, provided the structure of the system is a neural network.

## 4.2 Static Controller-Plant System

As we shall next show, the error gradient of a static controller-plant system takes the form of Equation 8. In static neural control applications, the system is composed of two components, the controller and plant, as shown in Figure 4. The controller is implemented by a multilayered neural network while the plant may be modeled by a neural network or a set of equations. When the plant is modeled using a neural network, the combination of the controller and plant form a static neural network. Therefore, the error gradient with respect to each weight of the neural controller takes the form of Equation 8 and can be calculated using the backpropagation algorithm.

If the plant is modeled by a set of equations, a differentiation of the plant equations with respect to the control vector in combination with a backpropagation can be used to compute the error gradient. The error of the system of Figure 4 is a function of the plant output, which is a function of the controller output. The controller output is a function of the weights of the network. Therefore, using the chain rule, the error gradient can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \lambda' \frac{\partial \mathbf{u}}{\partial w(i)} \tag{9}$$

where

$$\lambda' = \frac{\partial E}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{u}}.$$

Equations 8 and 9 have the same form. Because $\mathbf{u}$ is the output of a neural network, the error gradients with respect to the controller's weights are found by backpropagating $\lambda'$ through the neural network controller. The backpropagation term, $\lambda'$, is computed by multiplying $\partial E/\partial \mathbf{y}$ by the plant Jacobian matrix, $\partial \mathbf{y}/\partial \mathbf{u}$, which is calculated from the plant equations.

### 4.3 Training Neural Networks Implemented on Chip

It is also possible to train a static feedforward network which is implemented on a VLSI chip using first-order gradient descent. In this case, it is assumed that the precise mathematical equations of the neural network system are not known. Therefore, the derivatives cannot be calculated mathematically. Instead, they can computed by introducing small perturbations into the hidden layer nodes. The resulting perturbation at the output divided by the node perturbation approximates the output gradient with respect to the node. This method, in conjunction with LMS [9], can be used to calculate the output gradient with respect to the weights. This technique of calculating the gradient is known as Madaline Rule III (MR III) [10]. A VLSI chip manufactured by Intel supports this type of training [11].

The first-order gradient techniques discussed above are used not only for adapting static systems, they are also a key component of the rules used for adapting discrete-time dynamic systems. It is shown in the next section that these techniques are required for training dynamic systems composed fully or partially of neural networks.

## 5 Algorithms

In this section two basic types of algorithms, Euler-Lagrange based algorithms and recursive gradient update algorithms, both of which are used for training discrete-time dynamic systems, are discussed. The epochwise and on-line versions of both these types of algorithms are presented. As indicated by their name, the Euler-Lagrange algorithms are based on the discrete-time Euler-Lagrange equations [12]. These equations are used to calculate the error gradient with respect to the weights. The backpropagation-through-time algorithm, which is used for epochwise training of dynamic neural networks, is an example of an Euler-Lagrange based algorithm. The recursive gradient update algorithm is based upon a recursive equation for the output gradient which is derived from the dynamic system definition, Equation 1. The error gradient is easily computed using this output gradient. The recursive backpropagation algorithm, which is used for on-line training of dynamic neural networks, is an example of a recursive gradient update algorithm. The epochwise and on-line versions of algorithms based upon the Euler-Lagrange equations and the recursive gradient update equation can be used to train a variety of dynamic systems which contain neural networks as will be shown periodically in the remainder of this section.

Before deriving the Euler-Lagrange and recursive gradient update algorithm, it is worth mentioning that the stability of these algorithms cannot be guaranteed. Therefore, when using these algorithms, one must constantly monitor their performance. If instability becomes a problem when using one of these algorithms, it is often necessary to change certain parameters of the algorithm, often the learning rate, to overcome the problem.

For any given discrete-time dynamic system problem, either an Euler-Lagrange based algorithm or a recursive gradient update algorithm can be used to train the system. In Section 6, it is shown that the Euler-Lagrange based algorithm and recursive gradient update algorithm compute approximately the same error gradient for a given problem in both the epochwise and on-line case. Even though the algorithms are inherently equivalent, the computational and storage requirements of the algorithms are different. Therefore, the selection of the appropriate algorithm for a specific problem should be based upon the computational and storage requirements. These requirements are derived in Section 7.

### 5.1 An Algorithm Based on the Euler-Lagrange Equations

The discrete-time Euler-Lagrange equations in the calculus of variations provide a standard technique for calculating the first-order gradients of an error function. Using these equations, it is possible to calculate the epochwise gradient of any discrete-time dynamic system provided that the differentiability requirements on the system, discussed in Section 2, are met. When the dynamic system is composed fully of a feedforward neural network, the error gradient can be calculated using a combination of the Euler-Lagrange equations and the backpropagation algorithm. This combination is the basis of the backpropagation-through-time algorithm which was first introduced by Werbos [6]. A number of researchers including Nguyen and Widrow [3],

Pearlmutter [2] and Jordan [8] have successfully used the backpropagation-through-time algorithm to train dynamic networks.

In this section, the discrete-time Euler-Lagrange equations are first derived. Next, an epochwise training algorithm which uses these equations is discussed. Finally, training of dynamic systems composed fully or partially of neural networks using the Euler-Lagrange based algorithm is presented.

### 5.1.1 Discrete-Time Euler-Lagrange Equations

In order to use first-order gradient descent, we need to find the error gradient, $\partial^+ E/\partial w(i)$, for any given epoch. This gradient can be derived using the first chain rule expansion, Equation 2, and the following ordered set of equations, which are generated at each epoch.

$$\begin{aligned}
\mathbf{W}_0(i) &= \mathbf{W}(i) \\
\mathbf{y}_0 &= f(\mathbf{R}_0, \mathbf{Y}_0, \mathbf{W}_0(i)) \\
\mathbf{W}_1(i) &= \mathbf{W}(i) \\
\mathbf{y}_1 &= f(\mathbf{R}_1, \mathbf{Y}_1, \mathbf{W}_1(i)) \\
&\vdots \\
\mathbf{W}_k(i) &= \mathbf{W}(i) \\
\mathbf{y}_k &= f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(i)) \\
&\vdots \\
\mathbf{W}_{k_f}(i) &= \mathbf{W}(i) \\
\mathbf{y}_{k_f} &= f(\mathbf{R}_{k_f}, \mathbf{Y}_{k_f}, \mathbf{W}_{k_f}(i)) \\
E &= f(\mathbf{y}_0, \mathbf{y}_1, \ldots, \mathbf{y}_{k_f}, \mathbf{d}_0, \mathbf{d}_1, \ldots, \mathbf{d}_{k_f})
\end{aligned}$$

Using the first chain rule expansion for an ordered system, Equation 2, we can expand the ordered derivative, $\partial^+ E/\partial w(i)$, to obtain

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial E}{\partial w(i)} + \sum_{k=0}^{k_f} \left( \frac{\partial^+ E}{\partial \mathbf{y}_k} \frac{\partial \mathbf{y}_k}{\partial w(i)} + \frac{\partial^+ E}{\partial \mathbf{W}_k(i)} \frac{\partial \mathbf{W}_k(i)}{\partial w(i)} \right). \tag{10}$$

The terms $\partial E/\partial w(i)$ and $\partial \mathbf{y}_k/\partial w(i)$ are equal to zero because $E$ and $\mathbf{y}_k$ are not a direct function of $w(i)$ [1]. Thus, we find the expansion of Equation 10 can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial \mathbf{W}_k(i)} \frac{\partial \mathbf{W}_k(i)}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial^+ E}{\partial w_k(i)}. \tag{11}$$

We need to find an expression for the term $\partial^+ E/\partial w_k(i)$. This expression can be found by expanding the ordered derivative using Equation 2.

$$\frac{\partial^+ E}{\partial w_k(i)} = \frac{\partial E}{w_k(i)} + \sum_{j=k}^{k_f} \frac{\partial^+ E}{\partial \mathbf{y}_j} \frac{\mathbf{y}_j}{\partial w_k(i)} + \sum_{j=k+1}^{k_f} \frac{\partial^+ E}{\partial \mathbf{W}_j(i)} \frac{\partial \mathbf{W}_j(i)}{\partial w_k(i)}.$$

[1] In our definition of the partial derivative, all terms are held constant except the terms in the denominator of the partial derivative. Therefore, if the function which defines the numerator of the partial derivative does not contain the terms of the denominator directly, then the partial derivative is zero.

The terms $\partial E/\partial w_k(i)$ and $\partial \mathbf{W}_j(i)/\partial w_k(i)$ are equal to zero. The term $\partial \mathbf{y}_j/\partial w_k(i)$ is nonzero only when $k = j$. Using these results, we find the ordered derivative, $\partial^+ E/\partial w_k(i)$, to be

$$\frac{\partial^+ E}{\partial w_k(i)} = \frac{\partial^+ E}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial w_k(i)}. \tag{12}$$

Substituting Equation 12 into Equation 11, the error gradient is

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f}\frac{\partial^+ E}{\partial \mathbf{y}_k}\frac{\partial \mathbf{y}_k}{\partial w_k(i)} = \sum_{k=0}^{k_f}\lambda_k\frac{\partial \mathbf{y}_k}{\partial w_k(i)} \tag{13}$$

where

$$\lambda_k = \frac{\partial^+ E}{\partial \mathbf{y}_k}.$$

The term $\partial \mathbf{y}_k/\partial w_k(i)$ of Equation 13 is easy to calculate. The term $\partial^+ E/\partial \mathbf{y}_k$ must still be expanded. Once again, using the chain rule expansion, Equation 2, we expand, $\partial^+ E/\partial \mathbf{y}_k$, to find

$$\lambda_k = \frac{\partial^+ E}{\partial \mathbf{y}_k} = \frac{\partial E}{\partial \mathbf{y}_k} + \sum_{j=k+1}^{k_f}\left(\frac{\partial^+ E}{\partial \mathbf{y}_j}\frac{\partial \mathbf{y}_j}{\partial \mathbf{y}_k} + \frac{\partial^+ E}{\partial \mathbf{W}_j(i)}\frac{\partial \mathbf{W}_j(i)}{\partial \mathbf{y}_k}\right). \tag{14}$$

The term $\partial \mathbf{W}_j(i)/\partial \mathbf{y}_k$ is equal to zero. The term $\partial \mathbf{y}_j/\partial \mathbf{y}_k$ is also equal to zero when $j > k + L$, where $L$ is the maximum number of delays in the feedback of dynamic system. Using these results, Equation 14 can be written as

$$\lambda_k = \frac{\partial E}{\partial \mathbf{y}_k} + \sum_{j=1}^{L}\frac{\partial^+ E}{\partial \mathbf{y}_{k+j}}\frac{\partial \mathbf{y}_{k+j}}{\partial \mathbf{y}_k} \tag{15}$$

$$= \epsilon_k + \sum_{j=1}^{L}\lambda_{k+j}\frac{\partial \mathbf{y}_{k+j}}{\partial \mathbf{y}_k} \tag{16}$$

where

$$\epsilon_k = \frac{\partial E}{\partial \mathbf{y}_k}.$$

Equation 16 is a backward difference equation which can be solved using the following boundary conditions

$$\lambda_{k_f} = \epsilon_{k_f} \tag{17}$$

$$\lambda_{k_f+1}, \ldots, \lambda_{k_f+L} = 0. \tag{18}$$

We shall refer to equations 16, 17 and 18 as the Euler-Lagrange gradient equations. These gradient equations along with

$$\frac{\partial^+ E}{\partial w(i)} = 0$$

form the discrete-time Euler-Lagrange equations. Equation 19 guarantees that a solution of these equations results in either a minimum or maximum. It usually is not possible to find the analytic solution of these equations. Instead, numerical methods, such as first-order gradient descent, are used to search for an approximate solution.

### 5.1.2 Implementation of the Algorithm

Having derived the Euler-Lagrange equations, we introduce the Euler-Lagrange based algorithm which is used to calculated the error gradient with respect to the weights for an epoch. First, the discrete-time dynamic equation of the system, Equation 1, is iterated forward in time from iteration 0 to $k_f$. An appropriately selected training set element is used to supply the boundary conditions, $\mathbf{R}_0, \mathbf{r}_0, \ldots, \mathbf{r}_{k_f}$ and $\mathbf{Y}_0$, for the forward iteration of Equation 1. Next, the error gradients with respect to the outputs, $\lambda_k$, are calculated by backward iterating Equation 16 using the boundary conditions of equations 17 and 18. Finally, the results of the backward sweep are used to compute the error gradient, Equation 13.

The technique for calculating the error gradient presented above is independent of the dynamic system. As long as the output gradient with respect to the weights and recurrent inputs exists, the epochwise error gradient of a discrete-time dynamic system with adaptive parameters may always be computed using this method.

### 5.1.3 Training Dynamic Systems Composed of Neural Networks

If the system function, $f$, is implemented by a feedforward neural network, an interesting observation can be made. The error gradient summation computation, equation 13, contains terms of the form $\lambda_k \partial \mathbf{y}_k / \partial w_k(i)$. It is shown in Section 3.2 that terms of this form can be calculated using a backpropagation provide that $\mathbf{y}_k$ is the output of a neural network. Therefore, in the dynamic network case, each term of Equation 13 can be computed by backpropagating the vector $\lambda_k$ through the dynamic network. Furthermore, the summation terms of the backward sweep calculation, Equation 16, are of the form $\lambda_{k+j} \partial \mathbf{y}_{k+j} / \partial \mathbf{y}_k$. Once again, in the dynamic network case, these terms may be calculated using the backpropagation algorithm. Although it may seem that a large number of backpropagations are required for each epoch to calculate the error gradient, only $k_f + 1$ backpropagations are needed. By backpropagation of the vector $\lambda_k$ at each iteration of the backward sweep and by storing the results of this backpropagation in memory, the minimum number of backpropagations can be achieved. This technique of calculating the error gradient is known as the backpropagation-through-time algorithm.

Neural network controllers can be designed using the Euler-Lagrange based algorithm. Systems with the neural network plant model may be trained using the backpropagation-through-time algorithm. If the plant model consists of a set of equations, the summation terms of both the backward sweep calculation, Equation 16, and error gradient computation, Equation 13, may be calculated by backpropagating the vector $\lambda'_k = \lambda_k \partial \mathbf{y}_k / \partial \mathbf{u}_k$ at each iteration of the backward sweep.

Finally, a combination of the Euler-Lagrange based algorithm and the MRIII algorithm can be used to train a discrete-time neural network which is implement on a VLSI chip. The summation terms of both the backward sweep calculation, Equation 16, and the error gradient summation computation, Equation 13, can be computed using the MRIII algorithm at each backward iteration of the Euler-Lagrange based algorithm.

## 5.2 An On-Line Algorithm Based on the Euler-Lagrange Equations

The Euler-Lagrange based algorithm is an epochwise training technique. In many applications, such as real-time filtering and adaptive control, it is necessary to allow on-line training. In these cases, an on-line version of the Euler-Lagrange based algorithm, which is introduced in this section, can be used.

Generally, in the on-line case, at each forward iteration of the dynamic system, $k'$, the error gradient is first calculated and the weights are updated based upon this calculation. Using the results of the previous section, the error gradient could be calculated by iterating Equation 16, which is repeated here,

$$\lambda_k = \epsilon_k + \sum_{j=1}^{L} \lambda_{k+j} \frac{\partial \mathbf{y}_{k+j}}{\partial \mathbf{y}_k} \tag{19}$$

backwards through time from iteration $k'$ to 0. Because it is common to use the mean squared error in the on-line case, the boundary conditions of Equation 19 would take the form

$$\lambda_{k'} = -(\mathbf{d}_{k'} - \mathbf{y}_{k'}) \tag{20}$$

$$\lambda_{k'+1}, \ldots, \lambda_{k'+L} = 0 \tag{21}$$

$$\epsilon_0, \ldots, \epsilon_k, \ldots, \epsilon_{k'-1} = 0. \tag{22}$$

Finally, the results of the backward sweep are summed to produce the on-line error gradient using

$$\frac{\partial^+ E_{k'}}{\partial w(k')} = \sum_{k=0}^{k'} \frac{\partial^+ E_{k'}}{\partial w_k(k')} = \sum_{k=0}^{k'} \lambda_k \frac{\partial \mathbf{y}_k}{\partial w_k(k')}$$

which is similar to Equation 13 of the previous section.

### 5.2.1 Problems Associated with On-Line Implementation

Two problems arise when one attempts to determine the on-line error gradient of a dynamic system in this manner. First, the number of iterations, $k'$, for most on-line applications quickly becomes large. Because the error gradient is calculated by iterating a difference equation backwards from $k'$ to 0, the number of computations required to calculate the gradient grows linearly with $k'$. Obviously, this technique of calculating the gradient quickly becomes computationally expensive. Instead of using an algorithm whose computations grows linearly with the current iteration count, it is better to use an algorithm whose computations remain constant and are independent of the current iteration count. This can be accomplished by iterating Equation 19 backwards through time a constant, $T$, number of iterations. Using this idea, the error gradient is calculated by first iterating Equation 16 backwards in time from iteration $k'$ to $k' - T$ using the appropriate boundary conditions. After this computation, the error gradient is calculated using

$$\frac{\partial^+ E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \frac{\partial^+ E_{k'}}{\partial w_k(k')} = \sum_{k=k'-T}^{k'} \lambda_k \frac{\partial \mathbf{y}_k}{\partial w_k(k')}. \tag{23}$$

Of course, the error gradient computed using this method is an approximation of the true gradient. An example will illustrate the nature of this approximation. Figure 5, shows the values of $\partial^+ E_{k'}/\partial w_k(k')$ for some given dynamic system. By summing only a portion of these terms, it can be observed in Figure 6 that the resulting approximate error gradient is a *windowed* version of the true gradient. Thus, the validity of the approximate gradient depends upon how much of the gradient lies outside of the window of length $T$.

The second problem with directly using the Euler-Lagrange equations in the on-line case results from the weight changes at each iteration. In the on-line case, the system difference equation is

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(k)) \tag{24}$$

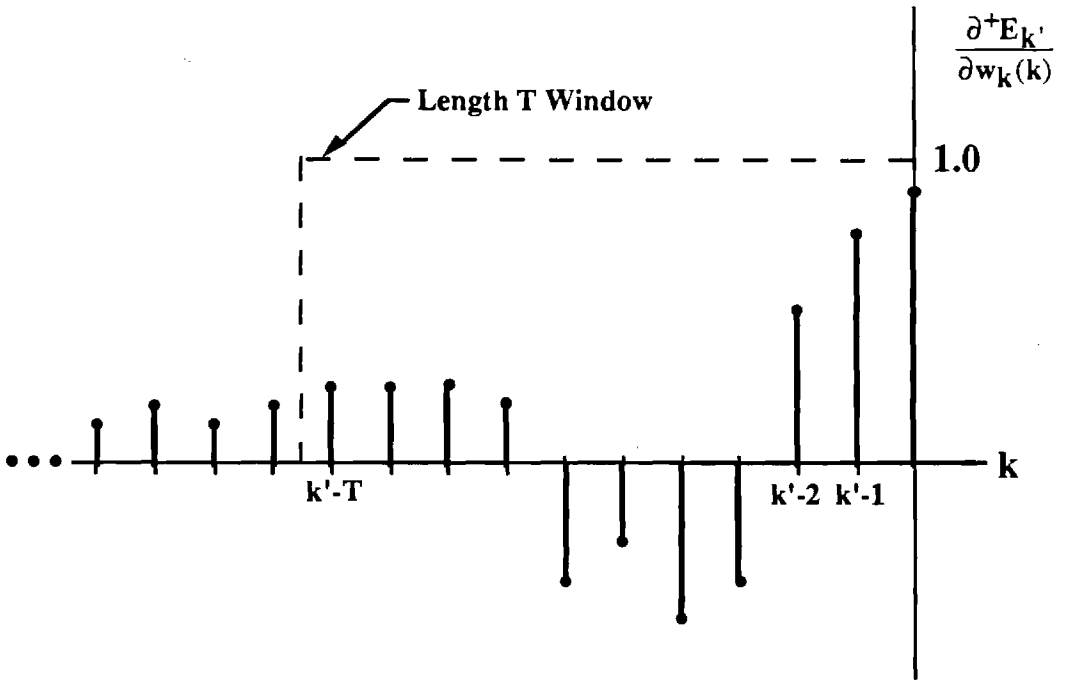


Figure 5: Error Gradients of a Typical System.

where the weight vector, $\mathbf{W}_k$, changes at each iteration according to Equation 7. The on-line error gradient, $\partial E_{k'}/\partial w(k')$, is defined with respect solely to the $k'^{th}$ weight vector. However, this weight vector is only used at iteration $k'$. Because the weight $w(k')$ only appears at iteration $k'$, one could conclude incorrectly that the error gradient could be calculated based solely upon iteration $k'$. This is incorrect because the weight vectors are related by Equation 7. In fact, in most cases of interest, the weights change slowly from iteration to iteration because the learning rate, $\mu$, of Equation 7 is small. Under this condition, we find

$$\mathbf{W}(k) \approx \mathbf{W}(k-1).$$

Using this approximation, the on-line error gradient summation of Equation 23, can be written as

$$\frac{\partial E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \frac{\partial^+ E_{k'}}{\partial w_k(k)} = \sum_{k=k'-T}^{k'} \lambda_k \frac{\partial \mathbf{y}_k}{\partial w_k(k)} \tag{25}$$

where $w_k(k)$ has replaced $w_k(k')$ in the partial derivative $\partial \mathbf{y}_k/\partial w_k(k)$. The combination of Equation 25 and a backward sweep of length $T+1$ allows the calculation of an approximate error gradient.

The error gradient approximation can be improved by exponentially weighting the terms of Equation 25. Because in most cases the weight change between $w(k)$ and $w(k')$ tends to become larger as $k$ is decreased starting from $k'$, it can be argued that the approximation $\partial E_{k'}/\partial w_k(k) \approx \partial E_{k'}/\partial w_k(k')$ becomes less valid as $k$ decreases. Under this assumption, when calculating the error gradient, the influence of the less accurate terms of Equation 25 should proportionally be reduced. This can be accomplished using the following exponential weighting scheme

$$\frac{\partial E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \alpha^{k'-k} \frac{\partial^+ E_{k'}}{\partial w_k(k)} = \sum_{k=k'-T}^{k'} \alpha^{k'-k} \lambda_k \frac{\partial \mathbf{y}_k}{\partial w_k(k)}. \tag{26}$$

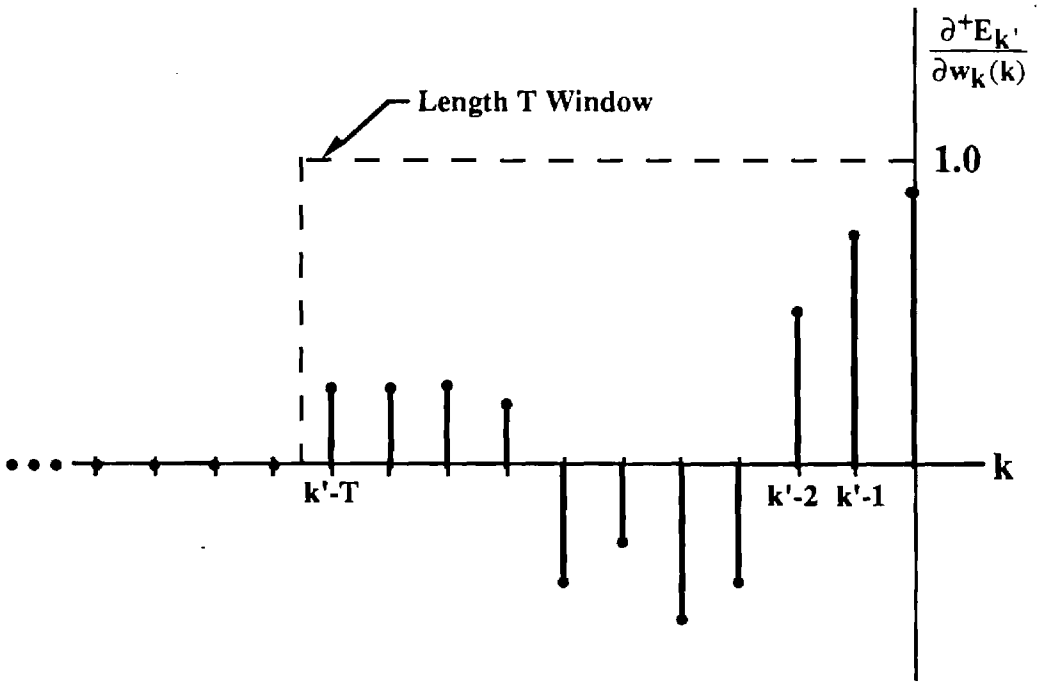


Figure 6: Windowed Error Gradients.

where the constant $0.0 < \alpha < 1.0$ is the weighting coefficient. The addition of exponential weighting causes the window to have the form shown in Figure 7. It should be noted, that exponential weighting may be desirable even in the epochwise training case if the backward sweep computation of Equation 16 is unstable.

### 5.2.2 Implementation of the Algorithm

Having derived approximate solutions for the two problems of implementing the Euler-Lagrange equations on-line, we can now present the on-line Euler-Lagrange based algorithm. The algorithm is based upon the following sequences being performed at each iteration $k'$: a forward propagation based on Equation 24, a backward sweep of length $T+1$ using Equation 19 and the boundary conditions of equations 20, 21, and 22, a calculation of the error gradient, Equation 26, and an update of the weights based upon Equation 7.

For a dynamic system with $Q$ weights, the on-line algorithm outlined above requires that $(T+1)Q$ weight vectors be stored. In most cases, this is an impractical amount of storage. Under these circumstances, an approximate output vector of the form

$$\mathbf{y}_k \approx f(\mathbf{R}_k, \mathbf{Y}_k, \mathbf{W}_k(k')) \tag{27}$$

may be used in Equation 19, the backward sweep equation, instead of the output vector defined in Equation 24. Equation 27 is a function of only one weight vector, $\mathbf{W}(k')$, therefore, only this vector needs to be stored. Using the approximation of Equation 27 results in a good approximation of $\partial E_{k'}/\partial w_k(k')$ provided that the weights change slowly.

In the on-line case, the summation terms of both the backward sweep calculation, Equation 19, and the error gradient computation, Equation 26, may be calculated in exactly the same manner as in the epochwise case, which is discussed in the last part of Section 5.1. Therefore, dynamic systems composed of a neural network or neural controller-plant system may be trained on-line. The on-line Euler-Lagrange algorithm can be used in conjunction with the backpropagation algorithm to calculate the error gradient of a dynamic neural network. This technique of calculating the on-line error is known as on-line backpropagation-through-time.

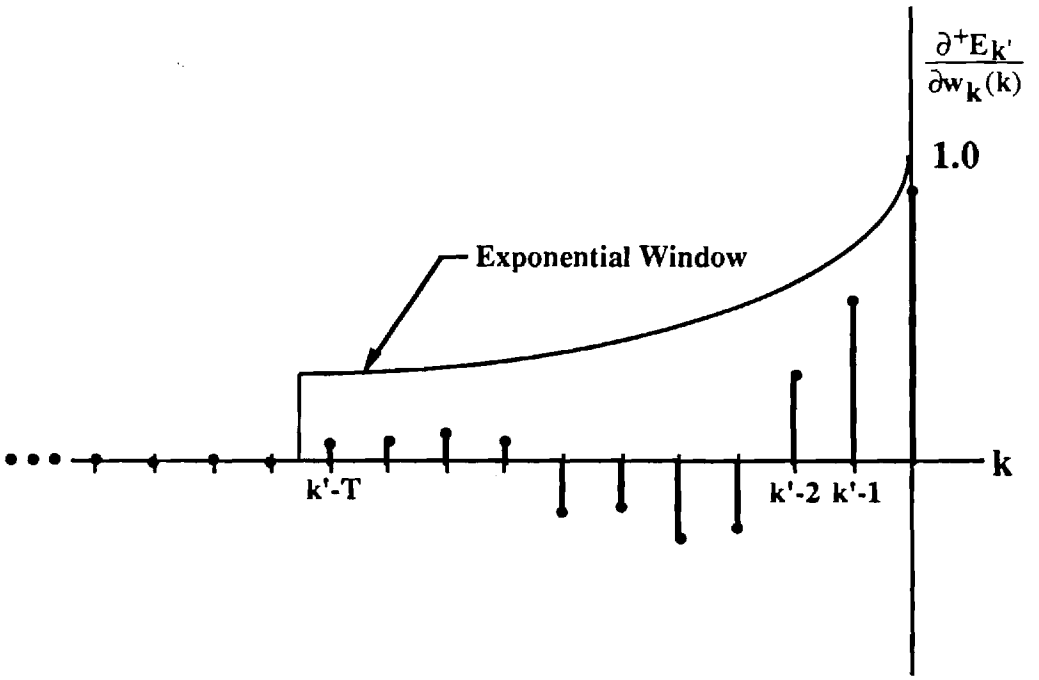


Figure 7: Exponentially Windowed Error Gradients.

One difficulty associated with the on-line Euler-Lagrange based algorithm is the selection of appropriate values for the constants $T$ and $\alpha$. Like the selection of the learning rate, $\mu$, there are no analytic procedures for choosing these constants. Instead, the selection should be based upon knowledge of the dynamic system, desired convergence rate and required misadjustment upon convergence.

## 5.3 Recursive Gradient Update Algorithm

The recursive gradient update algorithm provides yet another method for adapting a discrete-time dynamic system composed fully or partially of a neural network. The earliest version of the algorithm, which adapted a single linear node, was introduced by White in 1975 [13]. The algorithm received much attention during the later 70's in the adaptive signal processing community. It was found to suffer from stability problems and much of the recent research has been dedicated to overcoming this problem [14]. The recursive gradient update algorithm for nonlinear networks became well known because of a paper by Williams and Zipser [1]. Although this paper dealt only with single layer nonlinear networks, their version of the recursive algorithm could be generalized to multilayered networks by appropriatately selecting the connections between input and output. In this section, the recursive algorithm for a general system, which may be composed partially or fully of a neural network, is presented.

In both the epochwise and on-line cases, the recursive algorithm utilizes first-order gradient descent to minimize an appropriate error function. The difference between the Euler-Lagrange based algorithm and the recursive algorithm is the method in which the error gradient with respect to the weights is calculated. To gain an understand of the difference between the algorithms, the epochwise version of recursive algorithm is developed.

### 5.3.1 Epochwise Error Gradient

The Euler-Lagrange based algorithm is derived using the first chain rule expansion, Equation 2. In this section, the second chain rule expansion, Equation 3, is used to derive the recursive gradient update algorithm. Once again, we assume the error, $E$, is calculated using the ordered set of equations shown in Section 5.1.1.

We begin by expanding the error gradient using the second chain rule expansion, Equation 3, to obtain the following result.

$$\frac{\partial^+ E}{\partial w(i)} = \frac{\partial E}{\partial w(i)} + \sum_{k=0}^{k_f} \left( \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)} + \frac{\partial E}{\partial \mathbf{W}_k(i)} \frac{\partial^+ \mathbf{W}_k(i)}{\partial w(i)} \right) \tag{28}$$

The two terms, $\partial E/\partial w(i)$ and $\partial E/\partial \mathbf{W}_k(i)$, are equal to zero because the error, $E$, is not a direct function of $w(i)$ and $\mathbf{W}_k(i)$. Therefore, Equation 28 can be written as

$$\frac{\partial^+ E}{\partial w(i)} = \sum_{k=0}^{k_f} \frac{\partial E}{\partial \mathbf{y}_k} \frac{\partial^+ \mathbf{y}_k}{\partial w(i)}. \tag{29}$$

The first term of Equation 29, $\partial E/\partial \mathbf{y}_k$, is easy to compute. The second term, $\partial^+ \mathbf{y}_k/\partial w(i)$, can be found by using the second chain rule expansion once more.

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w(i)} + \sum_{j=0}^{k} \frac{\partial \mathbf{y}_k}{\partial \mathbf{W}_j(i)} \frac{\partial^+ \mathbf{W}_j(i)}{\partial w(i)} + \sum_{j=0}^{k-1} \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_j} \frac{\partial^+ \mathbf{y}_j}{\partial w(i)} \tag{30}$$

The term $\partial \mathbf{y}_k/\partial w(i)$ equals zero. The term, $\partial \mathbf{y}_k/\partial \mathbf{W}_j(i)$, of the first summation is nonzero only when $k = j$, therefore, this summation only contains one nonzero term. Furthermore, it is easy to verify that this term can be simplified to $\partial \mathbf{y}_k/\partial w_k(i)$. Finally, the first term of the second summation, $\partial \mathbf{y}_k/\partial \mathbf{y}_j$, is nonzero only when $k - j \leq L$. Using these results, Equation 30, can be written as

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w_k(i)} + \sum_{j=1}^{L} \frac{\partial \mathbf{y}_k}{\partial \mathbf{y}_{k-j}} \frac{\partial^+ \mathbf{y}_{k-j}}{\partial w(i)}. \tag{31}$$

The summation in Equation 31 can be eliminated using $\mathbf{Y}_k = [\mathbf{y}_{k-1}, \ldots, \mathbf{y}_{k-L}]^T$

$$\frac{\partial^+ \mathbf{y}_k}{\partial w(i)} = \frac{\partial \mathbf{y}_k}{\partial w_k(i)} + \frac{\partial \mathbf{y}_k}{\partial \mathbf{Y}_k} \frac{\partial^+ \mathbf{Y}_k}{\partial w(i)}. \tag{32}$$

This equation can be used to recursively calculate the output gradients for the entire epoch. It is usually initialized using

$$\frac{\partial^+ \mathbf{Y}_0}{\partial w(i)} = 0. \tag{33}$$

### 5.3.2 Implementation of the Algorithm

The epochwise error gradient can be calculated by first using Equation 32 to determine $\partial^+ \mathbf{y}_k/\partial w(i)$ for each iteration of the epoch. Once these ordered derivatives are determined, the error gradient can be calculated using Equation 29. In order to calculate the error gradient using this method, the output gradient at each iteration of the epoch must be available. If the outputs are stored in memory, a total of $(k_f + 1)NQ$ memory slots are required, where $N$ is the number of outputs and $Q$ is the number of weights. In many cases, this amount of memory may not be available. The memory requirements may be reduced to $(L + 1)NQ$ using a recursive calculation. Let $S_k$, an intermediate error gradient sum, be defined as

$$S_k = \sum_{j=0}^{k} \frac{\partial E}{\partial \mathbf{y}_j} \frac{\partial^+ \mathbf{y}_j}{\partial w(i)}.$$

Using the recursive equation

$$S_{k+1} = S_k + \frac{\partial E}{\partial \mathbf{y}_{k+1}} \frac{\partial^+ \mathbf{y}_{k+1}}{\partial w(i)} \tag{34}$$

which is initialized by

$$S_0 = \frac{\partial E}{\partial \mathbf{y}_0} \frac{\partial^+ \mathbf{y}_0}{\partial w(i)} \tag{35}$$

it follows

$$\frac{\partial^+ E}{\partial w(i)} = S_{k_f}.$$

The epochwise recursive update algorithm is implemented using the equations derived immediately above to calculate the error gradient. For a given epoch, the feedback input gradient, $\partial^+ \mathbf{Y}_0 / \partial w(i)$, is initialized using equation 33 while the intermediate error gradient sum, $S_k$, is initialized using Equation 35. At each iteration, from the initial iteration $k = 0$ to the final iteration $k = k_f$, the following sequence is performed: the system is forward propagated using the function $f$ defined in Equation 1, the output gradient is computed using the recursive calculation of Equation 32 and the intermediate error gradient is updated using Equation 34. Because $S_k$ and $\partial^+ \mathbf{y}_k / \partial w(i)$ are calculated at each iteration, only the previous $L$ output gradients need to be stored in memory, therefore, the storage requirements of the algorithm are approximately $(L+1)NQ$. After the final iteration, the error gradient is available as $S_{k_f}$.

In order to update the output gradient at each iteration using Equation 32, it is necessary to compute the direct output gradient, $\partial \mathbf{y}_k / \partial w_k(i)$, and the Jacobian matrix, $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$. Many different techniques can be used to calculate these terms depending upon the form of the dynamic system. As shown below, if the structure of the discrete-time dynamic system is a neural network, these two components can be found using $N$ backpropagations of $N$ appropriately selected vectors $\{\lambda_1, \ldots, \lambda_n, \ldots, \lambda_N\}$. These vectors are all backpropagated through the $k^{th}$ iteration of the dynamic network.

The $N$ vectors take the form

$$\lambda_{nj} = \begin{cases} 1 & \text{if } n = j \\ 0 & \text{otherwise} \end{cases}$$

where $\lambda_{nj}$ is the $j^{th}$ element of the row vector $\lambda_n \in R^{[1 \times N]}$. Each vector has only one nonzero component in the $n = j$ column. Because $\mathbf{y}_k$ is the output vector of a neural network, the backpropagation of the vector $\lambda_n$ through the network at iteration $k$ results in the calculation of the output gradient of the $n^{th}$ output, as indicated by

$$\lambda_n \frac{\partial \mathbf{y}_k}{\partial w_k(i)} = \frac{\partial y_k(n)}{\partial w_k(i)}.$$

Furthermore, as shown by,

$$\lambda_n \frac{\partial \mathbf{y}_k}{\partial \mathbf{Y}_k} = \frac{\partial y_k(n)}{\partial \mathbf{Y}_k}$$

backpropagating the $n^{th}$ vector, $\lambda_n$, back to the input nodes results in the calculation of the $n^{th}$ row of the Jacobian matrix. This shows that the direct output gradient, $\partial \mathbf{y}_k / \partial w_k(i)$, and Jacobian matrix, $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$, can be computed using $N$ backpropagations through the $k^{th}$ iteration of the dynamic network. Using this technique in the epochwise recursive gradient update algorithm shall be referred to as epochwise recursive backpropagation.

The computation of the direct output gradient and the Jacobian matrix for a neural controller-plant system is similar to the calculation for the dynamic neural network. In fact, if the plant model is a neural network, the technique discussed above can be used directly. If the plant model is based upon a set of equations, once again, the direct output gradient and Jacobian matrix can be computed using $N$ backpropagations through the neural network. In this case, the backpropagated vectors are of the form

$$\lambda'_n = \lambda_n \frac{\partial \mathbf{y}_k}{\partial \mathbf{u}_k} = \frac{\partial y_k(n)}{\partial \mathbf{u}_k}.$$

The backpropagation of $\lambda'$ through the neural controller of iteration $k$ results in the computation of the direct output gradient as indicated by

$$\lambda'_n \frac{\partial \mathbf{y}_k}{\partial w_k(i)} = \frac{\partial y_k(n)}{\partial \mathbf{u}_k} \frac{\partial \mathbf{u}_k}{\partial w_k(i)} = \frac{\partial y_k(n)}{\partial w_k(i)}.$$

Similarly, it can be shown that the $n^{th}$ row of the Jacobian matrix can be computed using the backpropagation of $\lambda'_n$ to the inputs of the neural controller.

If the system is composed of a neural network implemented in hardware, the two terms $\partial \mathbf{y}_k / \partial w_k(i)$ and $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$ of Equation 32 can be computed using MRIII. The MRIII algorithm uses the output gradient to calculate the error gradient. Therefore, this algorithm can be used without modification to find the two terms of Equation 32.

The epochwise recursive algorithm can be used to calculate the epochwise error gradient of any discrete-time dynamic system. In the next section, we show that this algorithm is easily extended to the on-line case.

## 5.4 On-Line Recursive Gradient Update Algorithm

The on-line version of the recursive gradient update algorithm is easily derived from the epochwise version. The calculation of the output gradient at each iteration performed by the epochwise version of the algorithm depends only upon the current and past values of the dynamic system. The lack of dependence on future values of the network in calculating the output gradient makes the algorithm attractive for on-line implementations.

In the on-line case, the mean squared error is often minimized by updating the weights at each iteration based upon an error gradient of the form

$$\frac{\partial E_{k'}}{\partial w(k')} = \frac{\partial E_{k'}}{\partial \mathbf{y}_{k'}} \frac{\partial^+ \mathbf{y}_{k'}}{\partial w(k')} \tag{36}$$

$$= -(\mathbf{d}_{k'} - \mathbf{y}_{k'})^T \frac{\partial^+ \mathbf{y}_{k'}}{\partial w(k')}. \tag{37}$$

The second term of Equation 37 may be calculated using the recursive gradient update calculation, Equation 32, which was derived in the previous section. Using this approach, the squared error is reduced at each iteration, and the mean squared error is approximately minimized.

The calculation of the output gradient using the recursive update computation,

$$\frac{\partial^+ \mathbf{y}_{k'}}{\partial w(k')} = \frac{\partial \mathbf{y}_{k'}}{\partial w_{k'}(k')} + \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{Y}_{k'}} \frac{\partial^+ \mathbf{Y}_{k'}}{\partial w(k')} \tag{38}$$

is based upon the assumption that the weights are constant. However, in the on-line case, the weights change from iteration to iteration according to Equation 7. In order to use the recursive equation, it is assumed that the weights change slowly or equivalently

$$\mathbf{W}(k) \approx \mathbf{W}(k-1).$$

This assumption allows

$$\frac{\partial^+ \mathbf{Y}_{k'}}{\partial w(k')} = \left[ \frac{\partial^+ \mathbf{y}_{k'-1}}{\partial w(k')}^T, \frac{\partial^+ \mathbf{y}_{k'-2}}{\partial w(k')}^T, \ldots, \frac{\partial^+ \mathbf{y}_{k'-L}}{\partial w(k')}^T \right]^T$$

of Equation 38 to be approximated as

$$\frac{\partial^+ \mathbf{Y}_{k'}}{\partial w(k')} \approx \left[ \frac{\partial^+ \mathbf{y}_{k'-1}}{\partial w(k'-1)}^T, \frac{\partial^+ \mathbf{y}_{k'-2}}{\partial w(k'-2)}^T, \ldots, \frac{\partial^+ \mathbf{y}_{k'-L}}{\partial w(k'-L)}^T \right]^T.$$

Because of this approximation, the on-line error gradient of Equation 37 can only be approximately calculated.

In a manner similar to the on-line Euler-Lagrange based algorithm, the accuracy of the error gradient approximation can be improved by introducing exponential weighting. The exponentially weighted output gradient can be calculated using

$$\frac{\partial \mathbf{y}_{k'}}{\partial w(k')} \approx \frac{\partial \mathbf{y}_{k'}}{\partial w_{k'}(k')} + \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{Y}_{k'}} \mathbf{F} \frac{\partial^+ \mathbf{Y}_{k'}}{\partial w(k')} \tag{39}$$

where the diagonal matrix $\mathbf{F} \in R^{[LN \times LN]}$ is used to perform the exponential weighting. In general, $\mathbf{F}$ takes the form

$$\mathbf{F} = \begin{bmatrix} \phi & 0 & . & . & 0 \\ 0 & \phi^2 & 0 & . & . \\ . & 0 & . & . & . \\ . & . & . & . & 0 \\ 0 & . & . & 0 & \phi^L \end{bmatrix}$$

where

$$\phi = \begin{bmatrix} \alpha & 0 & . & . & 0 \\ 0 & \alpha & 0 & . & . \\ . & 0 & . & . & . \\ . & . & . & . & 0 \\ 0 & . & . & 0 & \alpha \end{bmatrix} \tag{40}$$

with $\phi \in R^{[N \times N]}$ and $0.0 < \alpha < 1.0$.

The on-line recursive algorithm with exponential weighting is easily implemented. The feedback input gradient, which is used in the recursive gradient update equation, is initialize as indicated by Equation 33. The elements of the first weight vector, $\mathbf{W}(0)$, are randomly initialized. At each iteration, the following sequence is performed: a forward propagation implemented by Equation 24, a recursive update of the output gradient by Equation 39, a calculation of the error gradient which utilizes Equation 37 and an update of the weights using Equation 7.

The on-line recursive algorithm can be used to adapt the weights of a dynamic neural network or neural controller-plant system. As discussed in the previous section, in these cases, the two terms, $\partial \mathbf{y}_{k'}/\partial w_k(k')$ and $\partial \mathbf{y}_{k'}/\partial \mathbf{Y}_{k'}$, can be calculated using $N$ backpropagations. The combination of the on-line recursive algorithm and the backpropagation algorithm to adapt a dynamic network shall be referred to as the on-line recursive backpropagation algorithm. In addition, a neural network which is implemented on a VLSI chip can be adapted on-line using a combination of the on-line recursive algorithm and the MRIII algorithm. The on-line recursive algorithm provides an alternative technique to the on-line Euler-Lagrange based algorithm. A comparison of these algorithms is presented in the Section 7. This comparison will allow proper selection of an on-line algorithm for a given problem.

# 6 Comparison of the Algorithms

Although the Euler-Lagrange based algorithm and recursive gradient update algorithm may appear to be quite different, they both perform first-order gradient descent in weight space. In the epochwise case where no approximations are required to calculate the error gradient, the two algorithms are equivalent. In this case, because the selection of algorithm does not affect the convergence rate, the algorithm should be chosen on the basis of computational complexity and storage requirements which are discussed in Section 7.

In the on-line case, the two algorithms result in approximately identical weight updates given the same set of inputs. This can be shown using the formulation of Section 5. The exponentially weighted on-line error gradient of the Euler-Lagrange based algorithm, shown in Equation 26, is repeated here for comparison with the on-line gradient calculated by the recursive algorithm.

$$\frac{\partial^+ E_{k'}}{\partial w(k')} \approx \sum_{k=k'-T}^{k'} \alpha^{k'-k} \frac{\partial^+ E_{k'}}{\partial w_k(k)}. \tag{41}$$

An equation similar to Equation 41 can be derived for the on-line recursive algorithm. Using induction, it can be proved that the on-line recursive output gradient calculation of Section 5.4, Equation 39, is approximated as

$$\frac{\partial^+ \mathbf{y}_{k'}}{\partial w(k')} \approx \frac{\partial \mathbf{y}_{k'}}{\partial w_{k'}(k')} + \frac{\partial \mathbf{y}_{k'}}{\partial \mathbf{Y}_{k'}} \mathbf{F} \frac{\partial^+ \mathbf{Y}_{k'}}{\partial w(k')} = \sum_{k=0}^{k'} \alpha^{k'-k} \frac{\partial^+ \mathbf{y}_{k'}}{\partial w_k(k)}.$$

Substituting this result into the error gradient calculation of the on-line recursive algorithm, Equation 36, the weighted on-line error gradient calculated by the recursive algorithm is

$$\frac{\partial^+ E_{k'}}{\partial w(k')} \approx \frac{\partial E_{k'}}{\partial \mathbf{y}_{k'}} \sum_{k=0}^{k'} \alpha^{k'-k} \frac{\partial^+ \mathbf{y}_{k'}}{\partial w_k(k)} = \sum_{k=0}^{k'} \alpha^{k'-k} \frac{\partial^+ E_{k'}}{\partial w_k(k)}. \tag{42}$$

The two on-line techniques are approximately equivalent when equations 41 and 42 are approximately equal. This occurs when

$$\sum_{k=0}^{k'-T-1} \alpha^{k'-k} \frac{\partial E_{k'}}{\partial w_k(k)} \approx 0 \tag{43}$$

By appropriate selection of $\alpha$, one can guarantee that Equation 43 is made arbitrarily close to zero. Thus, we can conclude that the two on-line algorithms are approximately equivalent and that the validity of this approximation depends upon satisfying Equation 43.

# 7 Computational Complexity and Storage Requirements

If the two algorithms are equivalent in both the epochwise and on-line cases, then how does one choose which algorithm to use? Obviously, the choice should be based upon practical issues such as computational efficiency and storage requirements. In this section, the computational and storage requirements, with the system architecture, $f$, implemented by a neural network, are analyzed and compared. We choose to present only the dynamic neural network case for two reasons. First, the dynamic neural network system is the most common form of discrete-time dynamic system encountered in the neural networks field. Second, the computation and storage requirements are easy to calculate for any system once one understands these requirements for the dynamic neural network. Because we assume a neural network structure of the system architecture, the computational and storage requirements of the backpropagation-through-time, on-line backpropagation-through-time, epochwise recursive backpropagation and on-line recursive backpropagation algorithms are discussed below.

All four of these algorithms are based on the backpropagation algorithm. Therefore, an understanding of the computation requirements of this algorithm is necessary before deriving the complexity of the dynamic network algorithms. Epochwise training, using the backpropagation algorithm, consists of a forward propagation, a backward propagation and a weight update. Each of these computations requires on the order of $Q$ multiplications and additions, where $Q$ is the number of weights in the network. Therefore, the epochwise computational requirement of the backpropagation algorithm is on the order of $3Q$ multiplications and additions. Throughout the remainder of this section, we shall use the term operation to refer to a multiplication and addition.

## 7.1 Backpropagation-Through-Time Based Algorithms

The backpropagation-through-time algorithm, which is an epochwise technique of adapting a dynamic system, is based upon repeated forward and backward propagations through the dynamic network. For any given epoch, $k_f + 1$ forward and backward propagations are required. Because each of these propagations requires on the order of $Q$ multiplications and additions, $2(k_f + 1)Q$ operations are needed to calculate the error gradients, $\partial^+ E/\partial w_k(i)$, of each epoch. Using Equation 13 to compute the epochwise error gradient and Equation 5 to update the weights requires approximately $(k_f+1)Q$ operations. (In calculating the epochwise computational requirements, it is assumed that the weights are updated at each epoch.) The total number of multiplications and additions using the backpropagation-through-time algorithm, is

$$C_{BPTT} \approx 3(k_f + 1)Q. \tag{44}$$

The storage requirement of the backpropagation-through-time algorithm is derived from two primary components. First, the weights and their associated error gradients need to be stored in memory for efficient computation. These terms require $2Q$ floating point memory slots. Secondly, the output vector, $\mathbf{y}_k$ and external input vector, $r_k$ of the dynamic system at each iteration of the epoch are required for the calculation of the backward sweep. In order to minimize the storage requirements, only the inputs and outputs need be stored. The internal states of the system, such as the hidden node activation levels, can be recalculated from the input and output vectors. It should be noted, that minimization of the storage requirements may

increase the computation requirements by at most $(k_f + 1)Q$ because of the need to recalculate internal states. The external input is composed of $M$ floating point numbers while the output contains $N$ terms. Thus, $(k_f + 1)(M + N)$ memory slots are required for the external inputs and outputs of the dynamic system. Adding the two components together, the minimal storage requirement of the backpropagation-through-time algorithm is

$$S_{BPTT} \approx 2Q + (k_f + 1)(M + N).$$

The computational complexity of the on-line backpropagation-through-time algorithm is based upon the number of operations per iteration. These requirements are easily derived from those of the backpropagation-through-time algorithm. At each iteration, the error gradient is calculated using $T + 1$ backpropagations, and the weights are updated based upon this gradient. The backpropagations and weight updates require $2(T + 1)Q$ operations. In addition to these computations, one forward propagation of the dynamic system, which requires $Q$ multiplications and additions, is necessary. Using these calculations, the computation requirements per iteration of the on-line algorithm is

$$C_{OBPTT} \approx (2T + 3)Q.$$

In order to achieve the minimal storage requirements of the on-line backpropagation-through-time algorithm, the output vector, $\mathbf{y}_k$, must be defined as shown in Equation 27. In this case, only one weight vector need be stored in memory. In addition to the weights, the associated error gradients of each weight must be stored. Finally, only the output vector and external input vector of the previous $T + 1$ iteration are needed for calculation of the error gradient. Thus, the minimal storage requirement of the on-line backpropagation-through-time algorithm is

$$S_{OBPTT} \approx 2Q + (T + 1)(M + N).$$

## 7.2 Recursive Backpropagation Algorithms

The epochwise recursive backpropagation algorithm is based upon a forward propagation, a recursive update of the output gradient and a recursive update of the error gradient being performed at each iteration. The weights are updated at the final iteration based upon the error gradient. The computational complexity of the forward propagations is $(k_f + 1)Q$. The complexity of calculating the $k_f + 1$ output gradients is determined by the computational requirements of the recursive gradient update calculation, Equation 32. In the dynamic network case, the two terms $\partial \mathbf{y}_k / \partial w_k(i)$ and $\partial \mathbf{y}_k / \partial \mathbf{Y}_k$ are computed using $N$ backpropagations. In addition to the calculation of these two terms, a matrix-vector multiplication, which requires $N^2L$ operations, must be performed to compute the second term of Equation 32. Finally, the addition of the two terms of Equation 32 requires $N$ operations per weight. Adding all these components together, the computation complexity of the recursive output gradient calculation, Equation 32, is $(N^2L + 2N)Q$. The complexity of finding all output gradients is $(k_f + 1)(N^2L + 2N)Q$. At each iteration, the error gradient is updated using Equation 34 which requires a minimum of $N$ operations per weight. The computation requirements of updating the $Q$ error gradients over the $k_f + 1$ iterations of the epoch is $(k_f + 1)NQ$. Finally, $Q$ multiplications and additions are needed to update the weights. The computational requirements of the epochwise recursive backpropagation algorithm is

$$C_{RB} \approx (k_f + 1)(N^2L + 3N + 1)Q + Q. \tag{45}$$

As pointed out in Section 5.3.2, the storage requirements are determined by the recursive output gradient calculation, Equation 32. Therefore, the storage requirement of the epochwise recursive backpropagation algorithm is

| | Epochwise Algorithms | | On-Line Algorithms | |
|---|---|---|---|---|
| Requirements | Backpropagation-through-time | Recursive Backpropagation | Backpropagation-through-time | Recursive Backpropagation |
| Computational | $3(k_f+1)Q$ | $(k_f+1)(N^2L+3N+1)Q+Q$ | $(2T+3)Q$ | $(N^2L+N(L+3)+2)Q$ |
| Storage | $2Q+(k_f+1)(M+N)$ | $(L+1)NQ$ | $2Q+(T+1)(M+N)$ | $(L+1)NQ$ |

Table 1: Computation and Storage Requirements.

$$S_{RB} \approx (L+1)NQ.$$

The complexity of the on-line recursive backpropagation algorithm follows almost immediately from the computational requirements of the epochwise algorithm. At each iteration, the error gradient is calculated using an exponentially weighted recursive update calculation, Equation 39. The only difference between this equation and the one used in the epochwise case is the exponential weighting. Because the weighting constant, $\mathbf{F} \in R^{[LN \times LN]}$, is a diagonal matrix, $LN$ operations per weight are introduced by this matrix multiplication. Therefore, the recursive update calculation of Equation 39 requires $(N^2L + N(L+2))Q$ operations. The error gradient, Equation 37, is calculated using at least $NQ$ operations. Finally, the forward propagation of the system and the weight update both require $Q$ operations. The computational complexity per iteration of the on-line recursive backpropagation algorithm is

$$C_{ORB} \approx (N^2L + N(L+3) + 2)Q.$$

The only differences between the on-line and epochwise recursive algorithms are the exponential weighting of the recursive update equation and the weight update at each iteration of the on-line case. These two differences do not account for any difference in storage requirements. Therefore, the storage requirement of the on-line recursive backpropagation is

$$S_{ORB} \approx (L+1)NQ.$$

which is the same as that required for the epochwise recursive backpropagation algorithm.

## 7.3 Comparison of Algorithms

The computational and storage requirements are outlined in Table 1. Comparing the computational complexity of the two epochwise algorithms, equations 44 and 45, we find the epochwise backpropagation-through-time algorithm to be computationally more efficient than the epochwise recursive backpropagation algorithm. In general, epochwise algorithms based on the Euler-Lagrange equations are more efficient than those based on the recursive update equation. The computational inefficiency of the recursive technique is a result of the output gradient calculation which requires a matrix-vector multiplication for each weight. Although this calculation introduces inefficiency into the recursive backpropagation algorithm, it has the advantage of fixing the storage requirements to $(L+1)NQ$, which is independent of the number of iterations in an epoch. In some cases, where the total number of iterations, $k_f+1$, is large compared to the number of weights, $Q$, the epochwise recursive backpropagation algorithm may be advantageous to use for this reason. However, in many cases the total number of iterations is small compared to the number of weights and the storage requirements favor use of the epochwise backpropagation-through-time algorithm.

The ratio of the computational requirements of the two on-line algorithms

$$\frac{C_{OBPTT}}{C_{ORB}} \approx \frac{(2T+3)Q}{(N^2L+N(L+3)+2)Q} \tag{46}$$

can be used to compare the efficiencies of the two on-line algorithms. The most efficient on-line method can be chosen on the basis of the number of outputs, $N$, the maximum delay in the feedback loop, $L$, and the error gradient window length of the backpropagation-through-time algorithm, $T$. Using the ratio of Equation 46, a couple of general statements can be made about selection of an on-line algorithm based on computational efficiency. First, for systems with a small number of outputs, $N$, and a maximum delay, $L$, less than the window length, $T$, the on-line recursive algorithm is the most efficient technique of updating the weights. A general class of systems which meet these conditions are single output IIR adaptive filters. With this in mind, it is not surprising that in the adaptive filter field, the computationally more efficient on-line recursive technique has been well studied [9,14] while we are unaware of any attempts to use the on-line backpropagation-through-time algorithm. Secondly, for dynamic networks with a large number of outputs, $N$, or a large number of delays, $L$, the backpropagation-through-time algorithm is most efficient. Fully recurrent networks, which have a large number of outputs because each node is regarded as an output, should be adapted using the on-line backpropagation-through-time algorithm. The algorithm should also be used to train multidimension adaptive filters.

In addition to the computational efficiency, the storage requirements of the two on-line algorithms were derived in the previously in this section. On the basis of the storage requirements alone, the on-line backpropagation-through-time algorithm is preferable to the on-line recursive backpropagation algorithm when $(T+1)(N+M) < (L+1)NQ$. For almost all dynamic neural network systems, this inequality will hold, and the storage requirement will favor the on-line backpropagation-through-time algorithm.

# 8 Reducing On-Line Computational Complexity

Both the on-line backpropagation-through-time and on-line recursive backpropagation algorithm are computationally expensive. In this section, two techniques for reducing the number of computations are briefly presented.

## 8.1 Feeding Back the Desired Response

We have already stated that in the on-line case, it is common to minimize the square error at each iteration. In this case, a desired response vector, $\mathbf{d}_k$, must be available at each iteration. One method of speeding-up on-line learning, is to feed back the desired response instead of the output vector. Using this technique, the system equation is

$$\mathbf{y}_k = f(\mathbf{R}_k, \mathbf{D}_k, \mathbf{W}_k(k)) \tag{47}$$

where $\mathbf{D}_k = [\mathbf{d}_{k-1}^T, \mathbf{d}_{k-2}^T, \ldots, \mathbf{d}_{k-L}^T] \in R^{[NL\times 1]}$. Because the system defined by Equation 47 is independent of previous states of the system and therefore static, the error gradient can be calculated using a single backpropagation. Obviously, this technique is computationally less expensive than the two on-line algorithms of Section 5.

However, a price is to be paid for using this method. An approximate error gradient is calculated, whose validity depends upon the magnitude of the difference between the output and desire response vectors of the previous iterations. If these vectors are significantly different, a poor approximation of the error gradient is used to update the weights. Thus, even though the calculations per iteration are reduced, the number of iteration required to reach convergence will probably increase. Despite the increase in the number of iterations, feeding back the desired response is a method which can greatly decrease the computationally complexity per iteration.

Feeding back the desired response has been extensively studied in the field of adaptive signal processing [9]. This technique, which has been used to adapt single output linear filters, is known as the output-error formulation. A detailed analysis of the advantages and disadvantages of using this method for an adaptive linear filter can be found in Shynk, 1989 [14].

## 8.2 Redefining the On-Line Error Function

A second technique, which reduces the computational requirements while slightly increasing the time to convergence when using the on-line backpropagation-through-time algorithm, is based upon redefining the on-line error function. In the on-line case, the error function is commonly the squared error. In order to calculate the error gradient using the on-line backpropagation-through-time algorithm, $T$ backpropagations through the system are required at each iteration. However, by changing the error function, the computational complexity of the algorithm can be significantly reduced. Instead of using the squared error, the error function can be redefined as

$$\frac{\partial^+ E_{k'}}{\partial w(k')} = \begin{cases} \sum_{j=k'-C+1}^{k'} \frac{\partial^+ E_j}{\partial w(j)} & \text{if } (k' \bmod C) = 0 \\ 0 & \text{otherwise} \end{cases}$$

where $C$ is an integer constant greater than 1, and $k'$ denotes the forward iteration count. Using this definition, the error is nonzero every $C$ iterations. Thus, the error gradient need only be calculated every $C$ iterations instead of every iteration. For example, if $C = 10$, the number of computations is approximately reduced by a factor of 10, assuming the window length $T$ is not drastically increased. In general, the window length should be increased to $T + C$, and the learning rate, $\mu$, should be multiplied $C$. The reduction in computational requirements is accomplished by grouping the square error gradient calculations. Even though the number of computations is reduced, the number of iterations to convergence may increase because it may not be possible to multiply $\mu$ by a factor of $C$ for stability reasons. A more detailed explanation of this technique of reducing the on-line computations can be found in Williams and Peng, 1989 [15].

This method of speeding-up on-line learning can only be used for the on-line Euler-Lagrange based algorithm. Because the on-line recursive algorithm calculates the output gradients at each iteration, redefining the error gradient as shown in Equation 48 does not significantly change the computational complexity of the algorithm.

# 9 Examples

In this section, two examples which illustrate the uses of the dynamic system training algorithms are presented. The first example demonstrates the use of the algorithms for nonlinear controller design. A neural network is trained using the Euler-Lagrange based algorithm to provide the steering angle of a boat which is placed in a river with a nonlinear current. By providing the proper steering angle, the neural network guides the boat across the river to a designated dock position. The second example illustrates the use of the on-line recursive algorithm for adaptive filtering. In this example, an adaptive noise cancelling system is trained to eliminate filtered noise from a corrupted signal.

## 9.1 Nonlinear Control Example

In this example, a boat is initially placed in a river, which is 200 feet wide, within a region 100 feet upstream or downstream of a dock. The boat is powered by a constant thrust motor which is also used to point the boat in any desired direction. Starting from the initial position, it is desired to maneuver the boat to a dock, which is located on one shore of the river. Maneuvering the boat to the dock is made difficult by the stream's nonlinear current.

Let $x_k$ denote the distance from the center of the boat to the shore with the dock at iteration $k$. Let $y_k$ denote the distance of the center of the boat upstream or downstream of the dock. Assuming the current only to be a function of the distance from the shore, $x_k$, the equations of motion for the boat are

$$\begin{aligned} x_{k+1} &= x_k + 10cos(u_k) \\ y_{k+1} &= y_k + 10sin(u_k) + f_c(x_k). \end{aligned} \tag{48}$$

where $u_k$, the orientation of the boat given in radians, is the control signal, and $f_c(x_k)$, the influence of the current on the boat, is given in feet per iteration. The current, which is parabolic in nature with the greatest force in the middle of the stream at $x = 100$, is given by the following equation

$$f_c(x_k) = 7.5\left(\frac{x_k}{50} - \left(\frac{x_k}{100}\right)^2\right).$$

The control signal is supplied by the output of a three layer neural network. The first layer contains the two inputs, $x_k$ and $y_k$, which are the states of the system. The hidden layer contains ten sigmoidal neurons which are fully connected to the inputs and a bias. The output layer, which is linear, is fully connected to the hidden layer and the bias.

The boat system operates in an epochwise manner with the initial position determined randomly and the final position specified as the iteration prior to the boat hitting the dock's shore. For this reason, one of the two epochwise algorithms should be used to train the neural controller. Because of the computational efficiency of the epochwise Euler-Lagrange based algorithm, it was selected for training the controller. In order to make the boat come near to the dock at the final iteration, the following error function was used

$$E = (x_d - x_{k_f})^2 + (y_d - y_{k_f})^2$$

where $x_d$ is the $x$ position of the dock and $y_d$ is the $y$ position of the dock.

In order to train the neural controller, 4000 thousand training epochs were required with a learning rate, $\mu = 0.0001$. After training, four demonstration epochs, which are shown in Figure 8, were run. In the lower portion of Figure 8, the current is shown as a function of $x$. In order to show the boat graphically, it was necessary to move the two shores outward a distance equal to half the boat length. For this reason, the current near both shores is shown as zero. The four demonstration epochs show that by using the Euler-Lagrange based algorithm, it is possible to design a neural controller for the boat system.

## 9.2 Adaptive Filtering Example

In this example, an adaptive noise cancelling system was used to reduce additive noise from a corrupted signal. Before getting into the details of this example, the adaptive noise cancelling concept is introduced. Whenever an adaptive noise cancelling system is to be used, it is assumed that it possible to detect a noise source, $r_k$, which corrupts the original signal, $s_k$. Furthermore, it is assumed that a filter version of the noise, $n_k$, corrupts the original signal. Finally, it is assumed that the noise signal and the original signal are uncorrelated. The adaptive noise cancelling system receives as input the noise signal, $r_k$, and the corrupted signal, $s_k + n_k$. In order to eliminate the filtered noise from the corrupted signal, the noise signal is adaptively filtered and the result, $y_k$, is subtracted from the corrupted signal. If the adaptive filter is appropriately trained so that $y_k = n_k$, this subtraction will result in the output of the noise cancelling system, $\epsilon_k$, being equal to the original signal. Figure 9 shows an illustration of the basic noise cancelling system.

We have stated earlier that an on-line error function of the form $(d_k - y_k)^2$ minimizes $\mathbf{E}[(d_k - y_k)^2]$. For the adaptive noise cancelling system, we select an error function of the form $\epsilon_k^2$. Therefore, on-line adaptation of the system results in the minimization of $\mathbf{E}[\epsilon_k^2]$. We can find this quantity by expanding the expected values of $\epsilon_k$ as follows

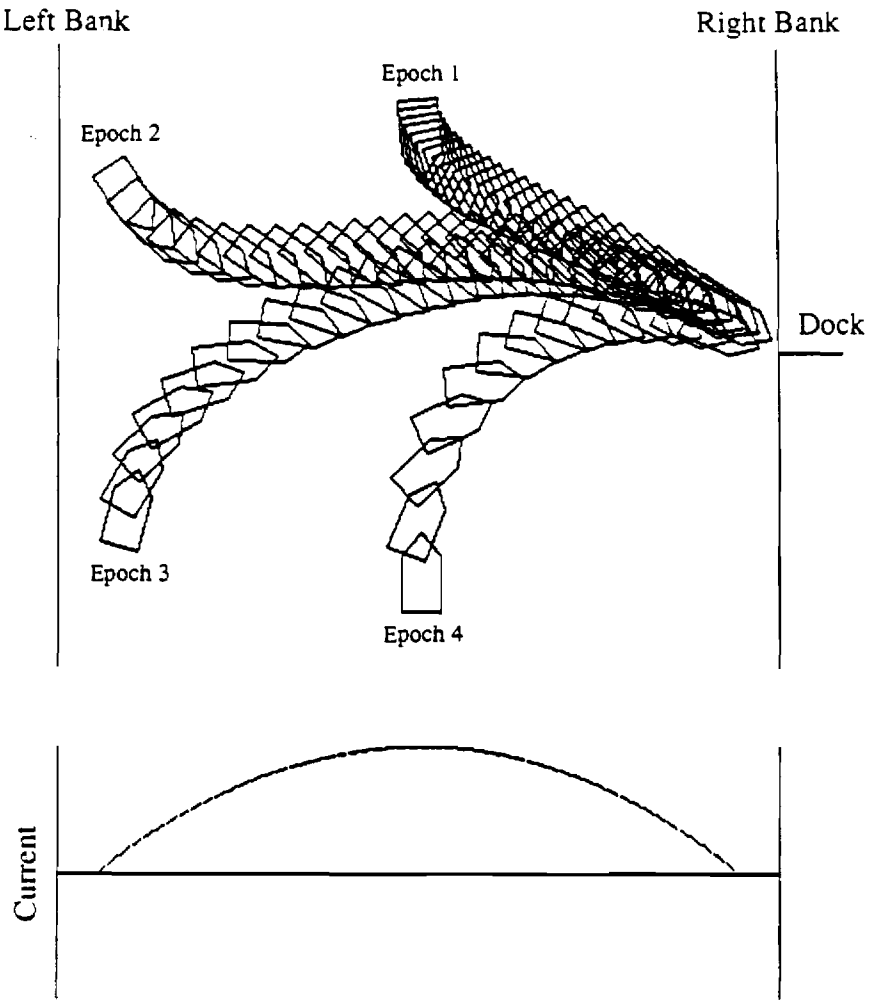


Figure 8: Nonlinear Control Example.

$$\begin{aligned} \mathbf{E}[\epsilon_k^2] &= \mathbf{E}[(s_k + n_k - y_k)^2] \quad (49) \\ &= \mathbf{E}[s_k^2 + 2s_k(n_k - y_k)^2 + (n_k - y_k)^2]. \quad (50) \end{aligned}$$

Assuming the original signal is uncorrelated with the noise signal and the adaptive filter output, Equation 50 can be written as

$$\mathbf{E}[\epsilon_k^2] = \mathbf{E}[s_k^2] + \mathbf{E}[(n_k - y_k)^2]. \quad (51)$$

Minimization of Equation 51 requires that $n_k = y_k$. Therefore, by using an on-line error function of the form $E_k = \epsilon_k^2$, the noise is adaptively eliminated from the corrupted signal by the adaptive noise cancelling system. For a more detail discussion of the adaptive noise cancelling concept, see Widrow and Stearns [9].

In our example, the original signal was

$$s_k = .25cos(.4k). \quad (52)$$

The noise signal, $r_k$, is selected randomly from a uniform distribution between -1.0 and 1.0. The filtered noise, $n_k$, is calculated using the following nonlinear difference equation

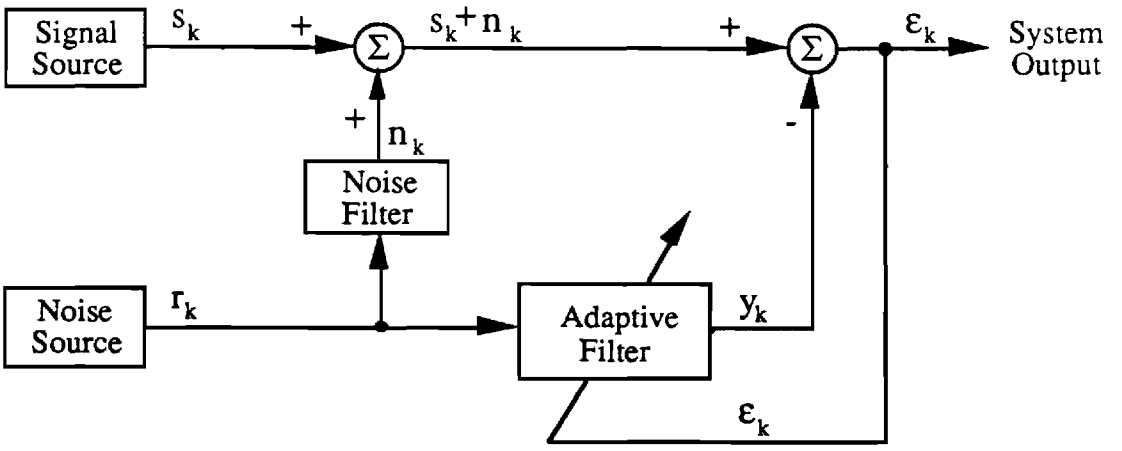


Figure 9: Adaptive Noise Cancelling System.

$$n_k = r_k + f_n(n_{k-1}) \tag{53}$$

where

$$f_n(n_{k-1}) = .5exp\left(\frac{-(n_{k-1}-1.0)^2}{0.67}\right) + .5exp\left(\frac{-(n_{k-1}+1.0)^2}{0.67}\right). \tag{54}$$

It should be noted that the noise filter contained nonlinear feedback.

The adaptive filter was implemented by a three layer feedforward neural network. The input layer was composed of two components, the noise signal, $r_k$, and the previous output of the adaptive filter, $y_{k-1}$. The hidden layer was composed of 17 hidden units each of which were squashed by the sigmoidal function. The first five nodes were connected through five different weights to the noise signal. The remaining ten nodes were connected to the feedback signal, $y_{k-1}$. In addition, each of the hidden units were connect to a bias. The output layer contained one linear unit which was connected to the hidden nodes and the bias through 18 separate weights.

One of the primary reasons for selecting the adaptive noise cancelling system as an example is that the feedback adaptive filter described above can only be trained using one of the on-line learning algorithms discussed in Section 5. The speed-up technique of feeding back the desired response cannot be used for this example because a desired response does not exist. The on-line recursive gradient update algorithm was selected for training the adaptive filter because it is computationally more efficient than the on-line Euler-Lagrange based algorithm when the number of outputs, $N$, and the number of delays, $L$, are both equal to 1.

A learning curve for the system, with the learn rate, $\mu = .005$, and the forgetting factor, $\alpha = .95$, is shown in Figure 10. The initial decrease in the mean squared error over the first couple hundred iterations is due to learning the feedforward component of the filter. The slow learning, which lasts for several thousand iterations, is due to learning the feedback component. The corrupted signal, $s_k + n_k$, and the original signal, $s_k$, for iterations 5900-6000 are shown in Figure 11. Notice that it is impossible to determine the characteristics of the original signal from the corrupted signal. The output signal, $\epsilon_k$, and original signal, $s_k$, for these same iterations are shown in Figure 12. Although the output signal is not perfect, the noise has been significantly reduced.

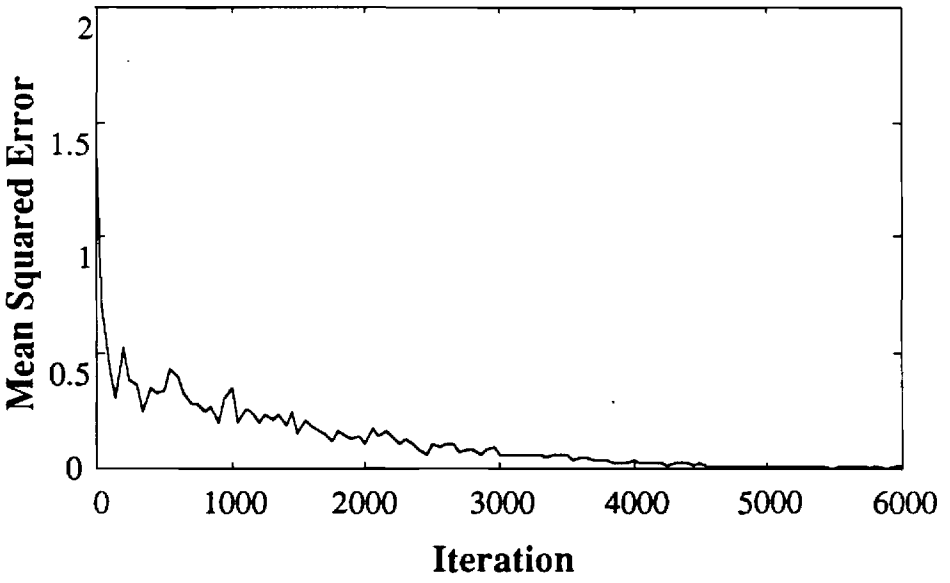


Figure 10: Learning Curve of the Noise Cancelling System.

# 10 Conclusion

The training of discrete-time dynamic systems using first-order gradient descent can be accomplished using either the Euler-Lagrange based algorithm or the recursive gradient update algorithm. Both these algorithms have been derived in this paper using the notation of the standard representation. Epochwise training can be accomplished using either of the two epochwise training algorithms which have been shown to produce identical weight updates. In general, because of both computational and storage requirements, the Euler-Lagrange based algorithm is preferable for epochwise training. However, the epochwise recursive algorithm may be desirable in cases where constant memory size is required. The two on-line algorithms produce approximately the same weight updates at each iteration. In general, the selection of an on-line algorithm is determined by the number of outputs, $N$, of the dynamic system. As this number increases, it becomes increasingly computationally expensive to use the recursive algorithm. Therefore, for large $N$, the Euler-Lagrange based algorithm is preferable for on-line training. Both on-line algorithms are computationally expensive. One method of reducing the computations is to feedback the desired responses, if they are available. Another method, which is applicable only to the Euler-Lagrange based technique, is to redefine the error function. Finally, two examples which illustrate the usefulness of the algorithms are presented. The first demonstrates the use of the Euler-Lagrange based algorithm for designing nonlinear state feedback controllers. The second illustrates the necessity of on-line algorithms in certain adaptive filtering problems.

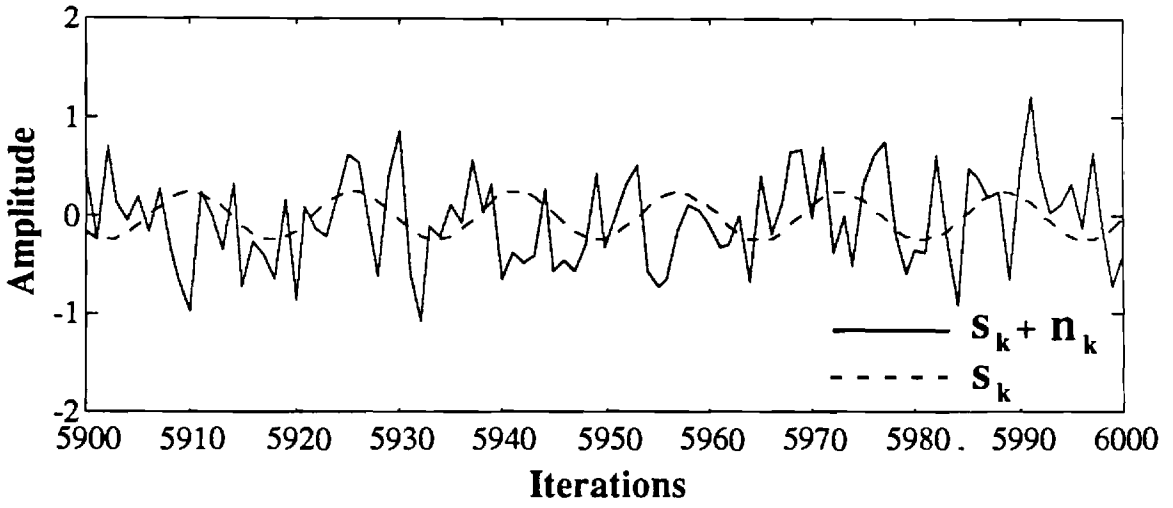


Figure 11: Corrupted Signal and Original Signal.

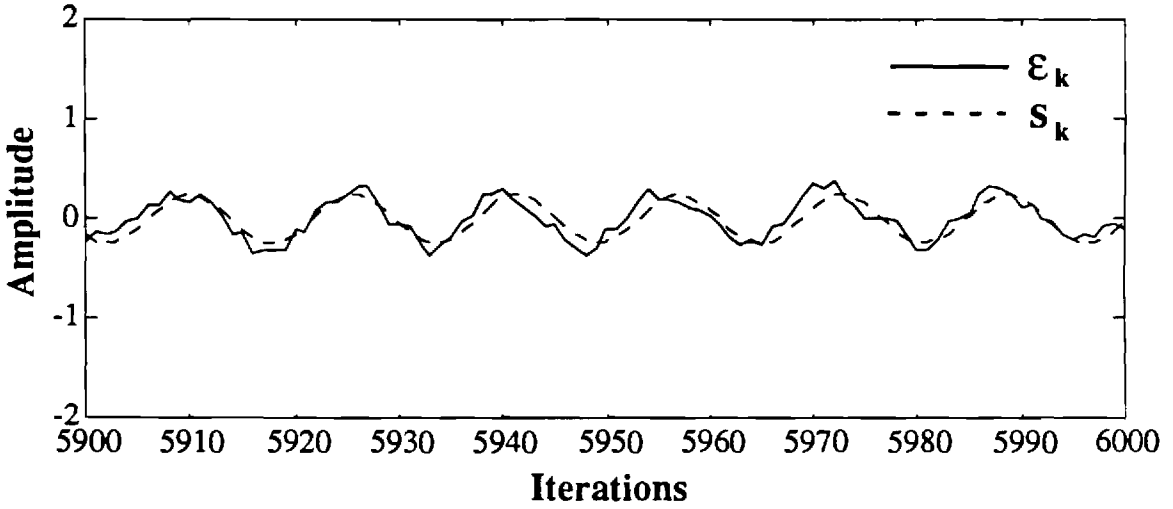


Figure 12: Output Signal and Original Signal.

# References


[1] R. J. Williams and D. Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, pages 270–277, Summer 1989.

[2] B. Pearlmutter. Learning state space trajectories in recurrent neuaral networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume II, pages 365–372, Washington, DC, June 1989.

[3] D. Nguyen and B. Widrow. Neural networks for self-learning control systems. *IEEE Control Systems Magazine*, April 1990.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In D. E. Rumelhart and J. L. McClelland, editors, *Parallel Distributed Processing*, volume 1, chapter 8. The MIT Press, Cambridge, MA, 1986.

[5] K. S. Narendra and K. Parthasarathy. Identification and control of dynamic systems using neural networks. *IEEE Transactions on Neural Networks*, pages 4–27, March 1990.

[6] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, Cambridge, MA, August 1974.

[7] R. J. Williams and D. Zipser. Gradient-based learning algorithms for recurrent connectionist networks. Technical Report NU-CCS-90-9, College of Computer Science, Northeastern University, Boston, MA 02115, April 1990.

[8] M. Jordan. Generic constraints on underspecified target trajectories. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pages 217–225, Washington, DC, June 1989.

[9] B. Widrow and S. D. Stearns. *Adaptive Signal Processing.* Prentice-Hall, Englewood Cliffs, NJ, 1985.

[10] D. Andes, B. Widrow, M. Lehr, and E. Wan. MRIII: A robust algorithm for training analog neural networks. In *Proceedings of the International Joint Conference on Neural Networks*, volume I, pages 533–536, Washington, DC, January 1990.

[11] M. Holler, *et al.* An electrically trainable artificial neural network (etann) with 10240 "floating gate" synapses. In *Proceedings of the International Joint Conference on Neural Networks*, volume II, pages 191–196, Washington, DC, June 1989.

[12] A. E. Bryson, Jr. and Y. Ho. *Applied Optimal Control.* Blaisdell Publishing Co., New York, 1969.

[13] S. A. White. An adaptive recursive digital filter. In *Proc. 9th Asilomar Conf. Circuits Syst. Compt.*, page 21, Nov. 1975.

[14] John J. Shynk. Adaptive IIR filtering. *IEEE ASSP Magazine*, April 1989.

[15] R. J. Williams and J. Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. Technical Report NU-CCS-90-10, College of Computer Science, Northeastern University, Boston, MA 02115 1990.